e-Informatica

software engineering journal

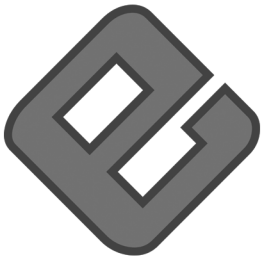
2010 volume 4 issue 1

e-Informatica

e-Informatica

software engineering journal

2010 | volume 4 | issue 1



e-Informatica

Wrocław University of Technology

Editors

Zbigniew Huzar (*Zbigniew.Huzar@pwr.wroc.pl*)
Lech Madeyski (*Lech.Madeyski@pwr.wroc.pl, http://madeyski.e-informatyka.pl/*)

Wrocław University of Technology
Institute of Applied Informatics
Wrocław University of Technology, 50-370 Wrocław, Poland

e-Informatica Software Engineering Journal
http://www.e-informatyka.pl/wiki/e-Informatica/

Printed in the camera ready form

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISSN 1897-7979

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Order No. 418/2010.

Editorial Board

Contents

Editorial

It is a pleasure to present to our readers the fourth issue of the e-Informatica Software Engineering Journal (ISEJ). The mission of the e-Informatica Software Engineering Journal is to be a prime international journal to publish research findings and IT industry experiences related to theory, practice and experimentation in software engineering. The scope of the journal includes methodologies, practices, architectures, technologies and tools used in processes along the software development lifecycle, but particular interest is in empirical evaluation.

The current issue of the journal includes eight papers. The first of the papers by Felkner and Sacha defines formal language that enables handling trust in distributed control systems. The sound and complete deductive system deriving credentials from initial credentials is presented and explained.

The second of the papers by Martenka and Walter is a contribution extending factor-strategy model proposed by Marinescu. It enables more comprehensive and traceable information concerning detected potential anomalies to the designer, resembling the human way of cognition.

The third of the papers by Pahl and Barrett presents a modelling and transformation technique for service-centric distributed systems. Authors capture behavioural aspects and associates quality of architectural structures at different levels of abstraction through patterns. Positive effect of the technique application is illustrated by a case study including design, maintenance and evolution of a system that has been developed by more than 20 people and maintained for more than ten years.

The objective of the fourth paper by Yu is to understand the changing patterns of software complexity. Common coupling is a measure of the system complexity but also it gives insight into software flexibility. How the coupling changes with the evolution of a software system is the subject of study on Apple Darwin, an open-source operating system.

The fifth paper by Hnatkowska and Kasprzyk proposes an approach to business logic implementation that enables easy response to business rules changes. Separation of business logic layer from business rule layer by introducing an integration layer is the core of the idea. The proof-of-concept implementation of the integration layer is presented in the aspect oriented language.

The sixth paper by Rychlý is an interesting application of Milner's π-calculus to describe behaviour of components in service-oriented architecture. A case study of the architecture for functional testing of complex safety-critical systems is presented.

The seventh paper by Staron and Meding presents methods for constructing prediction models of trends in defect inflow in large software projects. Two models are considered. The first one, so called short-term prediction model, is used to predict the number of defects discovered in the code up to three weeks in advance. The second one, long-term prediction model, provides the possibility of predicting the defect inflow for the whole project. The initial evaluation of these methods in a large software project at Ericsson shows that the models are sufficiently accurate and easy to deploy.

In the last paper Alhroob, Dahal and Hossain present a new technique of test cases generation extending the Integrated Classification Tree Methodology. The stress is put on extraction of legitimate test cases by removing the duplicate test cases and those incomputable with the software specifications. Large amounts

of time would have been needed to execute all of the test cases; therefore, a methodology is aimed to select the best testing path which guarantees the highest coverage of system units and avoids using all generated test cases.

We look forward to receiving quality contributions from researchers and practitioners in software engineering for the next issue of the journal.

Editors
Zbigniew Huzar
Lech Madeyski

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Deriving RT^T Credentials for Role-Based Trust Management

Anna Felkner*, Krzysztof Sacha**
*Research and Academic Computer Network
** Warsaw University of Technology
anna.felkner@gmail.com, k.sacha@ia.pw.edu.pl

Abstract

Role-based trust management languages define a formalism, which uses credentials to handle trust in decentralized, distributed access control systems. A credential provides information about the privileges of users and the security policies issued by one or more trusted authorities. The main topic of this paper is RT^T, a language which supports manifold roles and role-product operators to express threshold and separation of duties policies. The core part of the paper defines a relational, set-theoretic semantics for the language, and introduces a deductive system, in which credentials can be derived from an initial set of credentials using a set of inference rules. The soundness and the completeness of the deductive system with respect to the semantics of RT^T is proved.

1. Introduction

The problem of guaranteeing that confidential data and services offered by a computer system are not made available to unauthorized users is a challenging issue, which must be solved by reliable software technologies that are used for building high-integrity applications. The traditional solution to this problem is an implementation of some access control techniques, by which users are identified, and granted or denied access to a system data and other resources, depending on their individual or group identity. The examples of such solutions can be Mandatory Access Control (MAC) facilities, Discretionary Access Control (DAC) and Role-Based Access Control (RBAC) systems. Such an approach fits well into closed and centralized environments, in which the identity of users is known in advance.

Quite new challenges arise in decentralized and open systems, where the identity of users is not known in advance and the set of users can change. For example, consider a university, in which the students are enrolled and registered in particular faculties, and no central registry of all the students of that university exists. The policy of the university is such that a student is eligible to attend a lecture given by a faculty, regardless of the faculty in which he or she is actually registered. However, how could a faculty (the lecture owner) know that Peter Pan is eligible to attend the lecture, if his name is unknown to this faculty? The identity of the student itself does not help in making a decision whether he or she is eligible to attend or not. What is needed to make such a decision is information about the privileges assigned to Peter Pan by other authorities (is he registered in a faculty), as well as trust information about the authority itself (is the faculty a part of this university).

Trust-management system is a standardized solution for controlling security-critical services in high-integrity applications (Figure 1). It helps answer questions related to the conformance of potentially dangerous operations to a security policy of an organization, and provides the users with a language for writing the policies and controlling access to system services and

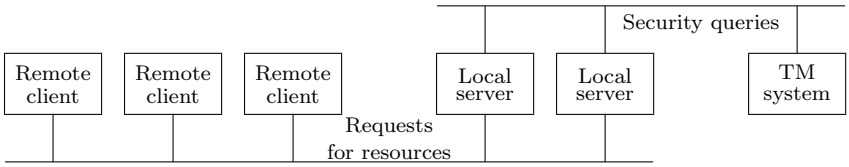


Figure 1. Trust management system

resources. The policies are no longer hard-coded into applications and therefore can be much easier to change. A designer of an application must only identify the security issues in the application and formulate appropriate queries to the trust-management system.

Such a conception of trust management, introduced in [2], has evolved since that time to a much broader context of assessing the reliability and developing trustworthiness for other systems and individuals [9]. In this paper, however, we will use the term trust management only in a meaning restricted to the field of access control.

The paper is organized as follows. An overview of the work related to role-based trust management systems and languages is given in Section 2. Section 4 describes the relational semantics of RT^T language. Section 6, which is the core part of our contribution, presents a deductive system, in which credentials can be derived from an initial set of credentials using a set of inference rules. A proof of the soundness and the completeness of the deductive system with respect to the semantics of RT^T is presented as well. Sections 3 and 5 provide the reader with illustrative examples. Final remarks and plans for future research are given in conclusions.

2. Related Work

Traditional access control systems usually rely on Role-Based Access Control model [14, 6, 7], which groups the access rights by the role name and limits the access to a resource to those users, who are assigned to a particular role. RBAC systems provide authorization decisions based on the identity of the users, and work well in centralized environment of an enterprise.

Trust management model represents quite another approach to access control, in which decisions are based on credentials (certificates) issued by multiple principals. A credential is an attestation of qualification, competence or authority, issued to an individual by a third party. Examples of credentials in real life include identification documents, social security cards, driver's licenses, membership cards, academic diplomas, certifications, security clearances, passwords and user names, keys, etc. A credential in a computer system can be a digitally signed document.

The potential and flexibility of trust management approach stems from the possibility of delegation: A principal may transfer limited authority over a resource to other principals. Such a delegation can be implemented by means of an appropriate credential. This way, a set of credentials can define the access control strategy and allow of deciding on who is authorized to access a resource, and who is not. A side-effect of delegation is such that a number of authorizing principals can be distributed over a network. A variety of problems arises if the credentials are stored in a decentralized manner.

The term trust management was first applied in the context of distributed access control in [2]. The first trust management system described in the literature was PolicyMaker [3], which defined a special assertion language capable of expressing policy statements, which were locally trusted, and credentials, which had to be signed using a private key. The next generation of trust management languages were KeyNote [1], which was an enhanced version of PolicyMaker, SPKI/SDSI [4] and a few other languages. All those languages allowed assigning privileges to entities and used credentials to delegate permissions from its issuer to its subject. What was missing in those languages was the possibility of delegation based on attributes of the entities and not on their identity.

Role-based trust management (RT) languages use roles to represent attributes [12]. The meaning of a role is a set of entities who have the attribute represented by the role. This meaning of roles captures the notion of groups of users in many systems and has been borrowed from Role-Based Access Control approach. The core language of RT family is RT_0, described in detail in [13]. It allows describing localized authorities for roles, role hierarchies, delegation of authority over roles and role intersections. All the subsequent languages add new features to RT_0.

RT_1 introduces parametrized roles, i.e. roles that are described using additional parameters, which can represent relationships between entities. RT_2 adds to RT_1 logical objects, which can represent permissions given to entities with respect to groups of logically related objects (resources). Those extensions can help in keeping the notation concise, but does not increase the expressive power of the language, because each combination of parameters in RT_1 and each permission to a logical object in RT_2 can be defined alternatively as a separate role in RT_0.

RT^T provides manifold roles and role-product operators, which can express threshold and separation of duties policies. A manifold role is a role that can be satisfied by a set of cooperating entities. A singleton role can be treated as a special case of a manifold role, whose set of cooperating entities is a singleton set. This way, RT_0 credentials can also be expressed in RT^T. A threshold policy requires a specified minimum number of entities to agree on some fact, e.g. in a requirement that two different bank cashiers must authorize a transaction. Separation of duties policy requires a set of entities, each of which fulfils a specific role, to agree before access is granted. Both types of policies mean that some transactions cannot be completed by a single entity, because no single entity has all the access rights required to complete the transaction.

RT^D provides mechanisms to describe delegation of role activations and selective use of role membership. This language is not covered in this paper. The features of RT^T and RT^D can be combined together with the features of RT_0, RT_1 or RT_2. A more detailed treatment of the role-based trust management family of languages can be found in [12].

2.1. The Language RT_0

Basic elements of all the RT languages are entities, role names, roles and credentials. **Entities** represent principals that can define roles and issue credentials, and requesters that can make requests to access resources. An entity can be identified by a user account in a computer system or by a public key. **Role names** represent permissions that can be issued by entities to other entities or groups of entities. **Roles** represent sets of entities that have permissions issued by particular issuers. A role is defined as a pair composed of an entity (role issuer) and a role name. **Credentials** define roles by pointing a new member of the role or by delegating authority to the members of other roles.

In this paper, we use nouns beginning with a capital letter or just capital letters, e.g. A, B, C, to denote entities and sets of entities. Role names are denoted as identifiers beginning with a small letter or just small letters, e.g. r, s, t. Roles take the form of an entity (the issuer of this role) followed by a role name separated by a dot, e.g. $A.r$. Credentials are statements in the language. A credential consists of a role, left arrow symbol and a role expression.

There are four types of credentials in RT_0, which should be interpreted in the following way:

- $A.r \leftarrow B$ – *simple membership*: Entity B is a member of role $A.r$.
- $A.r \leftarrow B.s$ – *simple inclusion*: Role $A.r$ includes (all members of) role $B.s$. This is a delegation of authority over r from A to B, because B may cause new entities to become members of the role $A.r$ by issuing credentials that define $B.s$.
- $A.r \leftarrow B.s.t$ – *linking inclusion*: Role $A.r$ includes role $C.t$ for each C, which is a member of role $B.s$. This is a delegation of authority from A to all the members of the role $B.s$. The expression $B.s.t$ is called a *linked role*.
- $A.r \leftarrow B.s \cap C.t$ – *intersection inclusion*: Role $A.r$ includes all the entities who are members

of both roles $B.s$ and $C.t$. This is a partial delegation from A to B and C. The expression $B.s \cap C.t$ is called an *intersection role.*

A formal, set-theoretic semantics of RT_0 has been defined in a slightly different manner in [13] and [8].

Let $\mathcal{E}$ be a set of entities, $\mathcal{R}$ a set of role names and $\mathcal{P}$ a set of RT_0 credentials. The semantics of the set $\mathcal{P}$ of RT_0 credentials is a function $\mathcal{S}_\mathcal{P}$:

$$\mathcal{S}_\mathcal{P} : \mathcal{E} \times \mathcal{R} \to 2^\mathcal{E}.$$

such that $\mathcal{S}_\mathcal{P}$ is the least fixpoint of the following sequence of functions R_i, which map roles to sets of entity names [8]:

1. R_0 maps each role to an empty set ϕ
2. $R_{i+1} = \bigoplus_{c \in \mathcal{P}} f(R_i, c)$

where $\bigoplus$ is the point-wise extension of a function and f is a function that, given a (partial) semantics R_i and a credential $A.r \leftarrow e$, returns all the entities that should be added to $R_i(A.r)$, as governed by e:

$$f(R_i, A.r \leftarrow B) = \{A.r \mapsto \{B\}\}$$
$$f(R_i, A.r \leftarrow B.s) = \{A.r \mapsto R_i(B.s)\}$$
$$f(R_i, A.r \leftarrow B.s.t) = \{A.r \mapsto \bigcup_{C \in R_i(B.s)} R_i(C.t)\}$$
$$f(R_i, A.r \leftarrow B.s \cap C.t) = \{A.r \mapsto R_i(B.s) \cap R_i(C.t)\}$$

2.2. The Language RT^T

At the syntax level, RT^T adopts all the four types of RT_0 credentials, and adds two new types of credentials. These are:

$A.r \leftarrow B.s \odot C.t$ – role $A.r$ includes one member of role $B.s$ and one member of role $C.t$. This allows expressing threshold policies.

$A.r \leftarrow B.s \otimes C.t$ – role $A.r$ includes one member of role $B.s$ and one member of role $C.t$, but those members of roles have to be different. This allows for expressing separation of duties policies.

The changes at the semantics level are greater, because the requesters as well as the issuers of RT^T credentials are no longer entities, but sets of entities, who can jointly fulfil a role. Such a change applies to all six types of credentials, also those, which are adopted from RT_0.

Formal definition of the semantics of RT^T is covered in Section 4.

3. Examples

The models discussed in this paper can be, in general, very complex. Therefore, we present here only simplified examples, with the intention to illustrate the basic notions and the notation. The first example demonstrates the use of RT_0 credentials, while the second one presents the use of RT^T credentials.

Example 1 (RT_0)

A person has the right to attend a *lecture*, given at a university U, when he or she is a *student* registered to a faculty of this university. To be able to fulfil the role of a $faculty$, an organization ought to be a *division* of the university and should conduct *research* activities. *John* is a student registered to F, which is a *division* of U, and which conducts *research* activities. The following credentials prove that *John* have the right to attend a *lecture*:

$$U.lecture \leftarrow U.faculty.student \quad (1)$$
$$U.faculty \leftarrow U.division \cap U.research \quad (2)$$
$$U.division \leftarrow F \quad (3)$$
$$U.research \leftarrow F \quad (4)$$
$$F.student \leftarrow John \quad (5)$$

Example 2 (RT^T)

The following example has been adopted from [11]. A bank B has three roles: *manager*, *cashier* and *auditor*. Security policy of the bank requires an *approval* of certain transactions from a *manager*, two *cashiers*, and an *auditor*. The two *cashiers* must be different. However, a *manager* who is also a *cashier* can serve as one of the two cashiers. The *auditor* must be different from the other parties in the transaction.

Such a policy can be described using the following credentials:

$$B.twoCashiers \leftarrow B.cashier \otimes B.cashier \quad (6)$$

$$B.managerCashiers \leftarrow B.manager \odot B.twoCashiers \quad (7)$$

$$B.approval \leftarrow B.auditor \otimes B.managerCashiers \quad (8)$$

Now, assume that the following credentials have been added:

$$B.cashier \leftarrow Mary \quad (9)$$

$$B.cashier \leftarrow Doris \quad (10)$$

$$B.cashier \leftarrow Alice \quad (11)$$

$$B.cashier \leftarrow Kate \quad (12)$$

$$B.manager \leftarrow Alice \quad (13)$$

$$B.auditor \leftarrow Kate \quad (14)$$

Then one can conclude that, according to the policy of B, the following sets of entities can cooperatively approve a transaction: $\{Mary, Doris, Alice, Kate\}$, $\{Mary, Alice, Kate\}$ and $\{Doris, Alice, Kate\}$.

4. The Semantics of RT^T

The syntax of a language defines language expressions, which are used to communicate information. The primary expressions of role-based trust management languages are credentials and sets of credentials, which are used as a means for defining roles.

The semantics of a language defines the meaning of expressions. Such a definition consists of two parts [10]: A semantic domain and a semantic mapping from the syntax to the semantic domain. The meaning of a language expression must be an element in the semantic domain.

The semantics of RT_0, which defines the meaning of a set of credentials as a function from the set of roles into the power set of entities, has no potential to describe the meaning of RT^T, which supports manifold roles and role-product operators. Therefore, we define in this section the meaning of a set of credentials as a relation over the set of roles and the power set of entities. Thus, we use a Cartesian product of the set of roles and the power set of entities as the semantic domain of a role-based trust management language. The semantic mapping would associate a specific relation between roles and entities with each set of credentials. Such a relational approach allows us to define a formal semantics of RT^T language [5].

Let $\mathcal{E}$ be the set of entities and $\mathcal{R}$ be the set of role names. $\mathcal{P}$ is a set of RT-credentials, which describe the assignment of sets of entities to roles, issued by other entities (or rather sets of entities).

The semantics of $\mathcal{P}$, denoted by $\mathcal{S}_\mathcal{P}$, is defined as a relation:

$$\mathcal{S}_\mathcal{P} \subseteq 2^\mathcal{E} \times \mathcal{R} \times 2^\mathcal{E},$$

An instance of this relation, e.g.: (A, r, X), maps the role $A.r$ to a set of entities $X \in 2^\mathcal{E}$. If the cardinality of set X is greater than one, then the role $A.r$ is a manifold role and the entities of set X must cooperate together in order to satisfy the role. The cardinality of set A can also be greater than one, which would mean that the role $A.r$ is governed jointly by the entities of set A.

If all the sets of entities are singleton sets, the semantics of RT^T reduces to the semantics of RT_0. This way, our definition covers all the RT languages including RT_0 through RT^T.

Denote the power set of entities by $\mathcal{F} = 2^\mathcal{E}$. Each element in $\mathcal{F}$ is a set of entities from $\mathcal{E}$ (a subset of $\mathcal{E}$). Each element in $2^\mathcal{F}$ is a set, compound of sets of entities from $\mathcal{E}$.

The semantics of $\mathcal{P}$ can now be described in an alternative way as a function:

$$\widetilde{\mathcal{S}}_\mathcal{P} : 2^\mathcal{E} \times \mathcal{R} \rightarrow 2^\mathcal{F}$$

which maps each role from $2^\mathcal{E} \times \mathcal{R}$ into a set of subsets of entities. The members of each subset must cooperate in order to satisfy the role.

Knowing the relation $\mathcal{S}_\mathcal{P}$, one can define the function $\widetilde{\mathcal{S}}_\mathcal{P}$ as follows:

$$\widetilde{\mathcal{S}}_\mathcal{P}(A, r) = \{X \in 2^\mathcal{E} : (A, r, X) \in \mathcal{S}_\mathcal{P}\}$$

The semantics of RT^T can now be defined formally in the following way.

Definition 1. *The semantics of a set $\mathcal{P}$ of RT^T credentials, denoted by $\mathcal{S}_\mathcal{P}$, is the smallest relation S_i, such that:*
1. $\mathcal{S}_0 = \phi$
2. $\mathcal{S}_{i+1} = \bigcup_{c\in\mathcal{P}} f(\mathcal{S}_i, c)$ for $i = 0, 1, \ldots$

which is closed with respect to function f, which describes the meaning of credentials in the following way (A, B, C, X, Y are sets of entities, may be singletons):

$$f(\mathcal{S}_i, A.r \leftarrow X) = \{(A, r, X)\} \quad (D_1)$$
$$f(\mathcal{S}_i, A.r \leftarrow B.s) = \{(A, r, X) : (B, s, X) \in \mathcal{S}_i\} \quad (D_2)$$
$$f(\mathcal{S}_i, A.r \leftarrow B.s.t) = \bigcup_{C:(B,s,C)\in\mathcal{S}_i}\{(A, r, X) : (C, t, X) \in \mathcal{S}_i\} \quad (D_3)$$
$$f(\mathcal{S}_i, A.r \leftarrow B.s \cap C.t) = \{(A, r, X) : (B, s, X) \in \mathcal{S}_i \wedge (C, t, X) \in \mathcal{S}_i\} \quad (D_4)$$
$$f(\mathcal{S}_i, A.r \leftarrow B.s \odot C.t) = \{(A, r, X \cup Y) : (B, s, X) \in \mathcal{S}_i \wedge (C, t, Y) \in \mathcal{S}_i\} \quad (D_5)$$
$$f(\mathcal{S}_i, A.r \leftarrow B.s \otimes C.t) = \{(A, r, X \cup Y) : (B, s, X) \in \mathcal{S}_i \wedge (C, t, Y) \in \mathcal{S}_i \wedge (X \cap Y) = \phi\} \quad (D_6)$$

5. Examples

We use the example sets of credentials from Section 3 to illustrate the definition of RT^T semantics.

Example 1 (RT_0)

The starting relation $\mathcal{S}_0$ is, by definition, empty. The sequence of steps to compute consecutive relations $\mathcal{S}_i$ can be described as follows:
$\mathcal{S}_0 = \phi$
$\mathcal{S}_1 = \{(\{U\}, division, \{F\}),\ (\{U\}, research, \{F\}),\ (\{F\}, student, \{John\})\}$
$\mathcal{S}_2 = \{(\{U\}, division, \{F\}),\ (\{U\}, research, \{F\}),\ (\{F\}, student, \{John\}),$
$(\{U\}, faculty, \{F\})\}$
$\mathcal{S}_3 = \{(\{U\}, division, \{F\}),\ (\{U\}, research, \{F\}),\ (\{F\}, student, \{John\}),$
$(\{U\}, faculty, \{F\}),\ (\{U\}, lecture, \{John\})\}$

The resulting relation $\mathcal{S}_3$ cannot be changed using the given set of credentials, hence: $\mathcal{S}_P = \mathcal{S}_3$. Because the RT language considered in this example is RT_0, all the sets of entities are singleton sets.

Example 2 (RT^T)

The sequence of steps to compute consecutive relations $\mathcal{S}_i$ starts from an empty set, $\mathcal{S}_0 = \phi$, and proceeds as follows. Credentials 9 through 14 are mapped in $\mathcal{S}_0$ into relation $\mathcal{S}_1$:
$\mathcal{S}_1 = \{(\{B\}, cashier, \{Mary\}),\ (\{B\}, cashier, \{Doris\}),$
$(\{B\}, cashier, \{Alice\}),\ (\{B\}, cashier, \{Kate\}),$
$(\{B\}, manager, \{Alice\}),\ (\{B\}, auditor, \{Kate\})\}$
Credential 6 adds the following instances to relation $\mathcal{S}_2$:
$\mathcal{S}_2 = \mathcal{S}_1 \cup \{$
$(\{B\}, twoCashiers, \{Mary, Doris\}),\ (\{B\}, twoCashiers, \{Mary, Alice\}),$
$(\{B\}, twoCashiers, \{Mary, Kate\}),\ (\{B\}, twoCashiers, \{Doris, Alice\}),$
$(\{B\}, twoCashiers, \{Doris, Kate\}),\ (\{B\}, twoCashiers, \{Alice, Kate\})\}$
Credentials 7 is resolved in $\mathcal{S}_3$:
$\mathcal{S}_3 = \mathcal{S}_2 \cup \{$
$(\{B\}, managerCashiers, \{Mary, Doris, Alice\}),$

$(\{B\}, managerCashiers, \{Mary, Alice\})$,
$(\{B\}, managerCashiers, \{Mary, Kate, Alice\})$,
$(\{B\}, managerCashiers, \{Doris, Alice\})$,
$(\{B\}, managerCashiers, \{Doris, Kate, Alice\})$,
$(\{B\}, managerCashiers, \{Alice, Kate\})\}$,

and credential 8 in $\mathcal{S}_4$:

$\mathcal{S}_4 = \mathcal{S}_3 \cup \{$
$(\{B\}, approval, \{Mary, Doris, Alice, Kate\})$,
$(\{B\}, approval, \{Mary, Alice, Kate\})$,
$(\{B\}, approval, \{Doris, Alice, Kate\})\}$,

The resulting relation $\mathcal{S}_4$ cannot be changed using the given set of credentials, hence: $\mathcal{S}_P = \mathcal{S}_4$. Because the RT language considered in this example is RT^T, there is a set of sets of entities assigned to each role.

6. Deductive system over RT^T credentials

RT^T credentials are used to define roles and roles are used to represent permissions. The semantics of a given set $\mathcal{P}$ of RT^T credentials defines for each role $A.r$ the set of entities which are members of this role. The member sets of roles can also be calculated in a more convenient way using a deductive system, which defines an operational semantics of RT^T language.

A **deductive system** consists of an initial set of formulae that are considered to be true, and a set of **inference rules**, that can be used to derive new formulae from the known ones.

Let $\mathcal{P}$ be a given set of RT^T credentials. The application of inference rules of the deductive system will create new credentials, derived from credentials of the set $\mathcal{P}$. A derived credential c will be denoted using a formula:

$$\mathcal{P} \succ c$$

which should be read: "credential c can be derived from a set of credentials $\mathcal{P}$".

Definition 2. *The initial set of formulae of a deductive system over a set $\mathcal{P}$ of RT^T credentials are all the formulae:*

$$c \in \mathcal{P}$$

for each credential c in $\mathcal{P}$. The inference rules of the system are the following:

$$\frac{c \in \mathcal{P}}{\mathcal{P} \succ c} \quad (W_1)$$

$$\frac{\mathcal{P} \succ A.r \leftarrow B.s \qquad \mathcal{P} \succ B.s \leftarrow X}{\mathcal{P} \succ A.r \leftarrow X} \quad (W_2)$$

$$\frac{\mathcal{P} \succ A.r \leftarrow B.s.t \qquad \mathcal{P} \succ B.s \leftarrow C \qquad \mathcal{P} \succ C.t \leftarrow X}{\mathcal{P} \succ A.r \leftarrow X} \quad (W_3)$$

$$\frac{\mathcal{P} \succ A.r \leftarrow B.s \cap C.t \qquad \mathcal{P} \succ B.s \leftarrow X \qquad \mathcal{P} \succ C.t \leftarrow X}{\mathcal{P} \succ A.r \leftarrow X} \quad (W_4)$$

$$\frac{\mathcal{P} \succ A.r \leftarrow B.s \odot C.t \qquad \mathcal{P} \succ B.s \leftarrow X \qquad \mathcal{P} \succ C.t \leftarrow Y}{\mathcal{P} \succ A.r \leftarrow X \cup Y} \quad (W_5)$$

$$\frac{\mathcal{P} \succ A.r \leftarrow B.s \otimes C.t \qquad \mathcal{P} \succ B.s \leftarrow X \qquad \mathcal{P} \succ C.t \leftarrow Y \qquad X \cap Y = \phi}{\mathcal{P} \succ A.r \leftarrow X \cup Y} \quad (W_6)$$

There could be a number of deductive systems defined over a given language. To be useful for practical purposes a deductive system must exhibit two properties. First, it should be sound, which means that the inference rules could derive only formulae that are valid with respect to the semantics of the language. Second, it should be complete, which means that each formula, which is valid according to the semantics, should be derivable in the system.

All the credentials, which can be derived in the system, either belong to set $\mathcal{P}$ (rule W_1) or are of the type: $\mathcal{P} \succ A.r \leftarrow X$ (rules W_2 through W_6). To prove the soundness of the deductive system, one must prove that for each new formula $\mathcal{P} \succ A.r \leftarrow X$, the triple (A, r, X) belongs to the semantics $\mathcal{S}_\mathcal{P}$ of the set $\mathcal{P}$.

Let we first note that all the formulae $\mathcal{P} \succ A.r \leftarrow X$, such that $A.r \leftarrow X \in P$ are sound. This is proved in Lemma 1.

Lemma 1. *If* $A.r \leftarrow X \in \mathcal{P}$ *then* $(A, r, X) \in \mathcal{S}_\mathcal{P}$.

Proof. The relation $\mathcal{S}_\mathcal{P}$, which defines the semantics of $\mathcal{P}$, is a limit of a monotonically increasing sequence of sets $S_0, S_1 \ldots$ such that $S_0 = \phi$. According to Definition 1: $f(S_0, A.r \leftarrow X) = (A, r, X)$ Hence, $(A, r, X) \in S_1$ and because $S_1 \subseteq \mathcal{S}_\mathcal{P}$ then $(A, r, X) \in S_P$. □

To prove the soundness of the deductive system over $\mathcal{P}$, we must prove the soundness of each formula $\mathcal{P} \succ A.r \leftarrow X$, which can be derived from the set $\mathcal{P}$. This is proved in Theorem 1.

Theorem 1. *If* $\mathcal{P} \succ A.r \leftarrow X$ *then* $(A, r, X) \in \mathcal{S}_\mathcal{P}$.

Proof. By induction with respect to the number n of inference steps, which are needed to derive a formula $\mathcal{P} \succ A.r \leftarrow X$.

If $n = 1$ then the formula $\mathcal{P} \succ A.r \leftarrow X$ could be derived only using rule W_1, because the premises of only this rule belong to the initial set of formulae of the deductive system. Hence, the thesis is true according to Lemma 1.

Consider $n > 1$ and assume for the inductive step that the thesis is true if the number of inference steps was not greater than n. We will show that it is true also in a case when the number of inference steps equals $n + 1$.

Each of the rules W_2 through W_6 could be used in the last $(n + 1)$ step of inference. All those five cases are discussed separately.

[W_2] The first premise of W_2 cannot be derived otherwise than using W_1. Hence, $A.r \leftarrow B.s \in P$. The second premise of $W_2 : \mathcal{P} \succ B.s \leftarrow X$ was derived from $\mathcal{P}$ using at most n steps of inference, hence, $(B, s, X) \in \mathcal{S}_\mathcal{P}$ according to the inductive hypothesis. By Definition 1, there exists such $\mathcal{S}_i$ that $(B, s, X) \in \mathcal{S}_i$, and $(A, r, X) \in f(\mathcal{S}_i, A.r \leftarrow B.s)$ according to (D_2). Because $f(\mathcal{S}_i, A.r \leftarrow B.s) \subseteq \mathcal{S}_{i+1} \subseteq \mathcal{S}_\mathcal{P}$ then $(A, r, X) \in \mathcal{S}_\mathcal{P}$.

[W_3] The first premise of W_3 cannot be derived otherwise than using W_1. Hence, $A.r \leftarrow B.s.t \in P$. The second premise of $W_3 : \mathcal{P} \succ B.s \leftarrow C$ was derived from $\mathcal{P}$ using at most n steps of inference, hence, $(B, s, C) \in \mathcal{S}_\mathcal{P}$ according to the inductive hypothesis. By Definition 1, there exists such $\mathcal{S}_i$ that $(B, s, C) \in \mathcal{S}_i$. Similarly, in the case of the third premise of $W_3 : \mathcal{P} \succ C.t \leftarrow X$, there exists such $\mathcal{S}_j$ that $(C, t, X) \in \mathcal{S}_j$. Let k be the maximum of (i, j). Then $(B, s, C) \in \mathcal{S}_k$ and $(C, t, X) \in \mathcal{S}_k$, and $(A, r, X) \in f(\mathcal{S}_k, A.r \leftarrow B.s.t)$ according to (D_3). Because $f(\mathcal{S}_k, A.r \leftarrow B.s.t) \subseteq \mathcal{S}_{k+1} \subseteq \mathcal{S}_\mathcal{P}$ then $(A, r, X) \in \mathcal{S}_\mathcal{P}$.

[W_4] The first premise of W_4 cannot be derived otherwise than using W_1. Hence, $A.r \leftarrow B.s \cap C.t \in P$. The second premise of $W_4 : \mathcal{P} \succ B.s \leftarrow X$ was derived from $\mathcal{P}$ using at most n steps of inference, hence, $(B, s, X) \in \mathcal{S}_\mathcal{P}$ according to the inductive hypothesis. By Definition 1, there exists such $\mathcal{S}_i$ that $(B, s, X) \in \mathcal{S}_i$. Similarly, in the case of the third premise of $W_4 : \mathcal{P} \succ C.t \leftarrow X$, there exists such $\mathcal{S}_j$ that $(C, t, X) \in \mathcal{S}_j$. Let k be the maximum of (i, j). Then $(B, s, X) \in \mathcal{S}_k$, $(C, t, X) \in \mathcal{S}_k$ and $(A, r, X) \in f(\mathcal{S}_k, A.r \leftarrow B.s \cap C.t)$ according to (D_4). Because $f(\mathcal{S}_k, A.r \leftarrow B.s \cap C.t) \subseteq \mathcal{S}_{k+1} \subseteq \mathcal{S}_\mathcal{P}$ then $(A, r, X) \in \mathcal{S}_\mathcal{P}$.

[W_5] The conclusion of W_5 is a formula $P \succ A.r \leftarrow X \odot Y$, which states that the set of entities that can play a role $A.r$ is a union of two another sets of entities X and Y. To prove the thesis we must show that $(A, r, X \cup Y) \in \mathcal{S}_\mathcal{P}$.

The first premise of W_5 cannot be derived otherwise than using W_1. Hence, $A.r \leftarrow B.s \odot$

$C.t \in \mathcal{P}$. Similarly as in case of W_4, the second and the third premises of W_5 were derived from $\mathcal{P}$ using at most n steps of inference. So, $(B, s, X) \in \mathcal{S}_\mathcal{P}$ and $(C, t, Y) \in \mathcal{S}_\mathcal{P}$. Then, there exists such k that $(B, s, X) \in \mathcal{S}_k$ and $(C, t, Y) \in \mathcal{S}_k$, and $(A, r, X \cup Y) \in f(\mathcal{S}_k, A.r \leftarrow B.s \odot C.t)$ according to (D_5). Because $f(\mathcal{S}_k, A.r \leftarrow B.s \odot C.t) \subseteq \mathcal{S}_{k+1} \subseteq \mathcal{S}_\mathcal{P}$ then $(A, r, X \cup Y) \in \mathcal{S}_\mathcal{P}$.

[W_6] The conclusion of W_6 is a formula $P \succ A.r \leftarrow X \otimes Y$, which states that the set of entities that can play a role $A.r$ is a union of two another sets of entities X and Y. To prove the thesis we must show that $(A, r, X \cup Y) \in \mathcal{S}_\mathcal{P}$.

The first premise of W_6 cannot be derived otherwise than using W_1. Hence, $A.r \leftarrow B.s \otimes C.t \in \mathcal{P}$. Similarly as in case of W_4, the second and the third premises of W_6 were derived from $\mathcal{P}$ using at most n steps of inference. So, $(B, s, X) \in \mathcal{S}_\mathcal{P}$ and $(C, t, Y) \in \mathcal{S}_\mathcal{P}$. Then, there exists such k that $(B, s, X) \in \mathcal{S}_k$ and $(C, t, Y) \in \mathcal{S}_k$. The fourth premise of W_6: $X \cap Y = \phi$, does not depend on the number of inference steps and is always true if W_6 could be applied. Hence, $(A, r, X \cup Y) \in f(\mathcal{S}_k, A.r \leftarrow B.s \otimes C.t)$according to (D_6). Because $f(\mathcal{S}_k, A.r \leftarrow B.s \otimes C.t) \subseteq \mathcal{S}_{k+1} \subseteq \mathcal{S}_\mathcal{P}$ then $(A, r, X \cup Y) \in \mathcal{S}_\mathcal{P}$. □

To prove the completeness of the deductive system over a set $\mathcal{P}$ of RT^T credentials, we must prove that a formula $P \succ A.r \leftarrow X$ can be derived using inference rules for each element $(A, r, X) \in \mathcal{S}_\mathcal{P}$. This is proved in Theorem 2.

Theorem 2. *If $(A, r, X) \in \mathcal{S}_\mathcal{P}$ then $\mathcal{P} \succ A.r \leftarrow X$.*

Proof. Assume $(A, r, X) \in \mathcal{S}_\mathcal{P}$. By Definition 1, there exists such $i \geq 0$ and such $c \in \mathcal{P}$ that $(A, r, X) \in f(\mathcal{S}_i, c)$. The proof of the thesis is by induction with respect to the value of index i.

If $i = 0$ then credential c must take the form of $A.r \leftarrow X$. This is because $\mathcal{S}_0 = \phi$ and $f(\mathcal{S}_0, d) = \phi$ for each credential d other than $A.r \leftarrow X$. Hence, $A.r \leftarrow X \in \mathcal{P}$ and the formula $\mathcal{P} \succ A.r \leftarrow X$ can be derived using rule W_1.

Let $i > 0$. Assume for the inductive step that the thesis is true, if the value of index i in the expression $(A, s, X) \in f(\mathcal{S}_i, c)$ was not greater than n. We will show that it is true also in the case when the value of index i equals $n + 1$.

Assume $(A, r, X) \in \mathcal{S}_\mathcal{P}$ and $(A, r, X) \in f(\mathcal{S}_{n+1}, c)$ for a certain $c \in \mathcal{P}$. The credential c can take one of the six forms allowed in RT^T. Each of these types of credentials will be discussed separately.

[$c = A.r \leftarrow X$] If this is the case, then the formula $\mathcal{P} \succ A.r \leftarrow X$ can be derived using rule W_1.

[$c = A.r \leftarrow B.s$] If $(A, r, X) \in f(\mathcal{S}_{n+1}, A.r \leftarrow B.s)$, then $(B, s, X) \in \mathcal{S}_{n+1}$ according to (D_2) of Definition 1. Hence, there exists a credential $c \in \mathcal{P}$ such that $(B, s, X) \in f(\mathcal{S}_n, c)$. This implies that $(B, s, X) \in \mathcal{S}_\mathcal{P}$ and $\mathcal{P} \succ B.s \leftarrow X$ according to the inductive hypothesis. Then $\mathcal{P} \succ A.r \leftarrow B.s$ and $\mathcal{P} \succ B.s \leftarrow X$, hence, $\mathcal{P} \succ A.r \leftarrow X$ is a conclusion of rule W_2.

[$c = A.r \leftarrow B.s.t$] If $(A, r, X) \in f(\mathcal{S}_{n+1}, A.r \leftarrow B.s.t)$ then according to (D_3) of Definition 1, there exists a set of entities C such that $(B, s, C) \in \mathcal{S}_{n+1}$ and $(C, t, X) \in \mathcal{S}_{n+1}$. Hence, there exists a credential $c_1 \in \mathcal{P}$ such that $(B, s, C) \in f(\mathcal{S}_n, c_1)$ and there exists a credential $c_2 \in \mathcal{P}$ such that $(C, t, X) \in f(\mathcal{S}_n, c2)$. This implies that $(B, s, C) \in \mathcal{S}_\mathcal{P}$ and $(C, t, X) \in \mathcal{S}_\mathcal{P}$, hence, $\mathcal{P} \succ B.s \leftarrow C$ and $\mathcal{P} \succ C.t \leftarrow X$ according to the inductive hypothesis. $\mathcal{P} \succ A.r \leftarrow X$ is a conclusion of rule W_3.

[$c = A.r \leftarrow B.s \cap C.t$] If $(A, r, X) \in f(\mathcal{S}_{n+1}, A.r \leftarrow B.s \cap C.t)$ then $(B, s, X) \in \mathcal{S}_{n+1}$ and $(C, t, X) \in \mathcal{S}_{n+1}$ according to (D_4) of Definition 1. Hence, there exist credentials c_1, c_2 such that $(B, s, X) \in f(\mathcal{S}_n, c_1)$ and $(C, t, X) \in f(\mathcal{S}_n, c_2)$. This implies that $(B, s, X) \in \mathcal{S}_\mathcal{P}$ and $(C, t, X) \in \mathcal{S}_\mathcal{P}$, hence, $\mathcal{P} \succ B.s \leftarrow X$ and $\mathcal{P} \succ C.t \leftarrow X$ according to the inductive hypothesis. $\mathcal{P} \succ A.r \leftarrow X$ is a conclusion of rule W_4.

[$c = A.r \leftarrow B.s \odot C.t$] If $(A, r, X) \in f(\mathcal{S}_{n+1}, A.r \leftarrow B.s \odot C.t)$, then according to (D_5) of Definition 1, there exist two sets of entities Z, Y such that $Z \cup Y = X$ and $(B, s, Z) \in \mathcal{S}_{n+1}$ and $(C, t, Y) \in \mathcal{S}_{n+1}$. Hence, there exist credentials c_1, c_2 such that $(B, s, Z) \in f(\mathcal{S}_n, c_1)$ and $(C, t, Y) \in f(\mathcal{S}_n, c2)$. This implies that $(B, s, Z) \in \mathcal{S}_\mathcal{P}$ and $(C, t, Y) \in \mathcal{S}_\mathcal{P}$, hence, $\mathcal{P} \succ B.s \leftarrow Z$ and $\mathcal{P} \succ C.t \leftarrow Y$ according to the inductive hypothesis. $\mathcal{P} \succ A.r \leftarrow X$ is a conclusion of rule W_5.

$[\boldsymbol{c = A.r \leftarrow B.s \otimes C.t}]$ If $(A, s, X) \in f(\mathcal{S}_{n+1}, A.r \leftarrow B.s \otimes C.t)$, then according to (D_6) of Definition 1, there exist two sets of entities Z, Y such that $Z \cup Y = X$ and $Z \cap Y = \phi$ and $(B, s, Z) \in \mathcal{S}_{n+1}$ and $(C, t, X) \in \mathcal{S}_{n+1}$. Hence, there exist credentials c_1, c_2 such that $(B, s, Z) \in f(\mathcal{S}_n, c_1)$ and $(C, t, Y) \in f(\mathcal{S}_n, c_2)$. This implies that $(B, s, Z) \in \mathcal{S}_{\mathcal{P}}$ and $(C, t, Y) \in \mathcal{S}_{\mathcal{P}}$, hence, $\mathcal{P} \succ B.s \leftarrow Z$ and $\mathcal{P} \succ C.t \leftarrow Y$ according to the inductive hypothesis. $\mathcal{P} \succ A.r \leftarrow X$ is a conclusion of rule W_6. □

A conclusion from Theorem 1 and Theorem 2 is such that the deductive system of Definition 2 is sound and complete with respect to the semantics of RT^T credentials. This way, the deductive system gives an operational definition of RT^T semantics.

7. Conclusions

This paper deals with modelling of trust management systems in decentralized and distributed environments. The modelling framework is a family of role-based trust management language RT^T. Two types of semantics for a set of RT^T credentials have been introduced in the paper.

A set-theoretic semantics of RT^T is defined as a relation over a set of roles and a power set (set of sets) of entities. All the members of a set of entities related to a role must cooperate in order to satisfy the role. This way, our definition covers the full potential of RT^T, which supports the notion of manifold roles and is able to express structure of threshold and separation-of-duty policies.

An operational semantics of RT^T is defined as a deductive system, in which credentials can be derived from an initial set of credentials using a set of inference rules. The semantics is given by the set of resulting credentials of the type $A.r \leftarrow X$, which explicitly show a mapping between roles and sets of entities.

The properties of soundness and completeness of the deductive system with respect to the semantics of RT^T are proved.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210. 1999.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Conference on Security and Privacy*, pages 164–173, 1996.

[3] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In *Financial Cryptography*, pages 1439–1456, 1998.

[4] D. Clarke, J. E. Elienb, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[5] A. Felkner and K. Sacha. The semantics of role-based trust management languages. In *Proc. Central and Eastern European Conference on Software Engineering Techniques CEE-SET*, pages 195–206, 2009.

[6] D. Ferraiolo and D. Kuhn. Role-based access control. In *Proc. 15th National Computer Security Conference*, pages 554–563, 1992.

[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[8] D. Gorla, M. Hennessy, and V. Sassone. Inferring dynamic credentials for role-based trust management. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, page 224, 2006.

[9] W. M. Grudzewski, I. K. Hejduk, A. Sankowska, and M. Wantuchowicz. *Trust Management in Virtual Work Environments: A Human Factors Perspective*. CRC Press, 2008.

[10] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stu. 2000.

[11] N. Li and J. Mitchell. RT: a role-based trust-management framework. In *Proc. 3rd DARPA Information Survivability Conference*

and Exposition, pages 201–212. IEEE Computer Society Press, 2003.

[12] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, pages 114–130, Oakland CA, 2002. IEEE Computer Society Press.

[13] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

[14] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Hierarchical Model for Evaluating Software Design Quality

Paweł Martenka*, Bartosz Walter*
*Institute of Computing Science, Poznań University of Technology
pawel.martenka@cs.put.poznan.pl, bartosz.walter@cs.put.poznan.pl

Abstract

Quality of software design has a decisive impact on several quality attributes of the resulting product. However, simple metrics, despite of their popularity, fail to deliver comprehensive information about the reasons of the anomalies and relation between them and metric values. More complex models that combine multiple metrics to detect a given anomaly are still only partially useful without proper interpretation. In the paper we propose a hierarchical model that extend the Factor-Strategy model defined by Marinescu in two ways: by embedding a new interpretation delivery mechanism into the model and extending the spectrum of data providing input to the model.

1. Introduction

Software design is considered one of the most complex human creative activities [13]. As such, the design process is prone to making errors, which significantly affect the quality of a software product resulting from the design. Therefore, there is a continuous search for models and approaches that could help both improving the design process and evaluating its quality.

Since software design is a quantifiable process, well-known code metrics are advocated as the primary solution for that problem. They are easy to compute, there is also plenty of experimental data showing the correlation between various metrics and desired quality attributes. However, metrics are just numbers, which often do not point to the design flaws, but rather provide rough and aggregate data. There are three main drawbacks of using the isolated metrics as direct providers of quality-related information:

1. There is no direct traceable connection between an actual cause and the value of a metric; usually it is the designer who is required to examine the values and identify the problem.
2. A vector of metric values has no meaning for the designer without a proper interpretation. Aggregate metrics are not subject to a straightforward interpretation.
3. Code metrics are unable to deliver complete information about software design. They need to be combined with diversed set of data to provide a more complete view.

Then, there is a need for more holistic approaches. One of them is a two-stage Factor-Strategy proposed by Marinescu ([17]), which is still based on metrics, but also addresses some of their weaknesses. It is a framework for building rule-based descriptions of design anomalies, which builds a navigable path between metrics and actual violations of high-level design principles. Unfortunately, this approach has also drawbacks. Such principles usually refer to abstract notions like cohesion or coupling, which still are not directly pointing to actual flaws. Moreover, actual code anomalies often result from multiple violations of different nature, for which the rules could be not properly

configured. For example, the Large Class bad smell [12], which describes classes bearing too much responsibility, typically denotes an overly complex, low-cohesive class with lots of members. Due to a large number of symptoms suggesting the presence of the flaw, metrics pointings to them must be combined and evaluated in non-linear and fuzzy manner to deliver an effective and useful measurement mechanism. Thus, the Factor-Strategy model, which is based on simple and strict rules, still does not provide a flexible abstraction for such flaws.

In this paper we propose a hierarchical model for evaluating design quality which is based on the Factor-Strategy concept, but extends it in several ways. It provides designers with hierarchical, custom-grained information, which helps in tracing the causes of flaws, and also enriches the spectrum of utilized sources of data.

The paper is structured as follows. Section 2 provides an overview of existing literature and approaches used for similar problems. In Section 3 we present Factor-Strategy model in a more detailed way, and in Section 4 we propose the hierarchical model. Section 5 contains a simple exemplary instance of the model, along with early experimental evaluation results. Section 6 summarizes our findings and proposes further extensions to the model.

2. Related Work

Historically, first attempts to quantitatively evaluate the design quality of object-oriented software were directly derived from code metrics. Metric suites proposed by Chidamber and Kemerer [6], e Abreu [9] and others were designed to capture the most important internal characteristics of object oriented software, like cohesion and coupling, and the use of mechanisms embedded in the object paradigm. A strong evidence has been collected pointing to correlation between these metrics and external quality characteristics.

These characteristics were further investigated by Briand et al. [3, 2], who noted that they are too ambiguous to be effectively captured by generalized, aggregate metrics. As an effect, they proposed several specific metrics, which analysed different flavours of cohesion and coupling.

Some researchers went in the opposite direction, building more holistic approaches to modelling design anomalies. Beck, the author of eXtreme Programming methodology, coined a term of "code bad smell" for a general label for describing structures in the code that suggest for the possibility of refactoring [11]. Since specific smells describe anomalies that can result from many initial causes, they should also be backed by several symptoms [23], e.g. diversed sets of metrics. Moonen et al. [22] proposed a method for automating smell detection based on analysis of source code abstract syntax trees. Kothari et al. in [16] defined a framework for building tools that perform partially automated code inspections and transformations.

Dhambri et al. in [8] proceeded a step further and employed visualisation techniques for detecting anomalies. The main idea was based on presenting some software quality attributes (e.g. measured by metrics) to a software design expert, who made the final decision. Another work, by Simon and Lewerentz [21], focused on refactorings driven by distance based cohesion. Distance between members of classes (fields and methods) was visualised in a 3D space, so that an expert could decide on appropriate assignment of class members and possibly suggest refactorings.

Based on critics of the simplistic metric-based quality models, Marinescu proposed Factor-Strategy model [17], composed of two stages: detection strategies stage responsible for identifying an anomaly, and composition stage that evaluates the impact of suspects found in the previous step on the high-level quality factors.

This model was further extended. Ratiu [20] encapsulated the detection strategies with a new model which incorporated code changes history into the classification mechanism. The new model has two main advantages:

1. removes false positives from the detected suspects set,
2. emphasizes the most harmful suspects.

Similar concept – use of historical data – was also exploited by Graves et al. [14] and Khoshgoftaar et al. [15]. Graves presented a few models to predict fault incidence and Khoshgoftaar introduced a regression model to predict software reliability, both based on the code history.

3. The Factor-Strategy Model

As Marinescu noted, classical models of design quality evaluation do not provide explicit mapping between metrics and quality criteria, so the rules behind quality quantification are implicit and informal. The metrics-based models can provide information about existence of a problem, but they do not reveal the actual cause of a problem. Hence, there is a need for a more comprehensive and holistic model.

The Factor-Strategy model has been proposed as a response to the above-mentioned weaknesses. It is composed of two main elements: the Detection Strategy and the composition step.

The Detection Strategy (DS) is defined as a quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code.

Rules are configured by a set of selected and suitable metrics. In consequence, DS provides a more abstract level of interpretation than individual metrics do, so that the numeric values of these metrics do not need to be interpreted in isolation.

Metrics are combined into rules using two basic mechanisms: filtering and composition. Filters transform metrics values whereas the composition operators aggregate into a rule. Marinescu gives a following example of a Detection Strategy instance for the Feature Envy smell:

> FeatureEnvy := ((AID, HigherThan(4))
> and (AID, TopValues(10%))
> and (ALD, LowerThan(3)) and (NIC, LowerThan(3))

This examplary rule uses three metrics: Access of Import-Data (AID), Access of Local Data (ALD) and Number of Import Classes (NIC) processed with *HigherThan*, *TopValues* and *LowerThan* filters, and composed with *and* composition operator.

Application of DS on a set of software entities (e.g. classes) results in:

1. a set of detected suspects,
2. a vector of metrics values for each suspect.

Using this data, a score for a DS is calculated and mapped to a normalised value (a ranked score). The score can be interpreted as a higher-level metric for the strategy. Marinescu provides a few exemplary formulas for computing the score, for example the simplest is the number of suspects for a given DS.

Quantification of high-level quality factors is based on an aggregation of ranked strategies and rules. Formulas for aggregation can vary from a simple mean value, where DS and the rules have equal weight, to more sophisticated, weighted methods. Selection of a method for aggregation depends on the measurement goals. The aggregated value – which is a score for the quality factor, is also mapped to the ranked score to provide qualitative information (labelled ranked scores).

4. Hierarchical Model

The Factor-Strategy model overcomes major problems of the classical solutions but still has a few drawbacks. The first doubt refers to the completeness of strategies suite: they need to be configured for every anomaly, so even the biggest set of strategies does not cover all possible flaws.

The second weakness is concerned about limiting the data sources to metrics only. As noted in [23], anomalies typically require multi-criteria detecting mechanisms, including data from dynamic execution, configuration management repository, analysis of Abstract Syntax Tree patterns etc. Ratiu and others [20, 14, 15] proved usefulness of historical data for quality evaluation. Van Emden [22] and Baxter [1] presented examples how Abstract Syntax Trees (ASTs) could be exploited as a source of quality-related data. The extended spectrum of sensor types, embedded into Factor-Strategy model, may improve its sensitivity, accuracy and correctness.

The final remark refers to the fact that operators used for defining detection rules are strict, ie. they define a borderline, which may classify very similar entities to different categories. Provided that the borderline is set up arbitrary, it can significantly affect the results of evaluation.

The goal of this research is to develop a hierarchical model which tackles the mentioned problems and weaknesses. It extends the Factor-Strategy model mainly in two areas:

1. diversed data sources are used instead of metrics only,
2. a simple mechanism for dealing with fuzzy problems is proposed.

4.1. Structure of the Model

The structure of the hierarchical model and its relation to the Factor-Strategy approach is shown on Fig. 1. At the top of the model there are high-level quality criteria (or characteristics), which are combined with detected lower-level patterns and rules violations. Pattern and rule detection methods are supported by data coming from various data sources, e.g. metrics, historical data, results of dynamic behaviour and abstract syntax trees (AST), which improves accuracy of the detection mechanism.

The model schema shows a hierarchy of elements, but also a hierarchy of information. The evaluation criteria provide the most abstract and the most aggregated information. A designer can track down the hierarchy to get more detailed information and find the cause of a problem indicated by the criteria.

4.2. Analysis of Detection Rules and Design Principles

Detection strategies, which are the core part of the original Factor-Strategy model, are configurable sets of rules aiming at capturing violations of the well-known principles of design, based on quantified data. However, actual design anomalies present in code do not always match the predicted and configured set of strategies. They can also violate multiple principles concurrently or – on the other hand – remain ignored by existing strategies.

The analysis mechanism present in the hierarchical model can be divided into three parts:

1. new data selection approach,
2. metrics quantisation,
3. entity-level aggregation.

4.2.1. Data Selection

Classical quality models employ a set of selected metrics for evaluation of quality factor (or factors). For example, a model presented by Briand et al. in [4] is built upon metrics which are supposed to measure coupling, inheritance, polymorphism and size, and is oriented on fault-proneness prediction. Also instances of Detection Strategies in [17] consist of diverse sets of metrics.

The model presented in this section promotes different approach. Typically, behind every principle of software design an internal quality characteristic is present. Based on this observation, the selection of metrics should be strictly oriented on such characteristic. On the other hand, the selected metrics should be simple, suitable and adequate in the context of measured characteristic. As a consequence, some types of metrics should be avoided:

1. strongly aggregating measures, like COF (Coupling Factor defined by Abreu et al. in [9]), which are biased by compensation problem – some parts of highly-coupled design can be masked by parts which are loosely-coupled,
2. metrics which are ambiguously defined, or those capturing ambiguous concepts; Khaled El-Emam in [10] argues that the notion of cohesion is too general to provide significant results,
3. metrics which try to capture multiple characteristics at a time or appear not related to the expected characteristic, eg. Basili et al. in [5] argue that *WMC* metric actually measures software size instead of complexity.

Following the postulate of diversed data sources, the model creation process should incorporate as many sources as is needed to

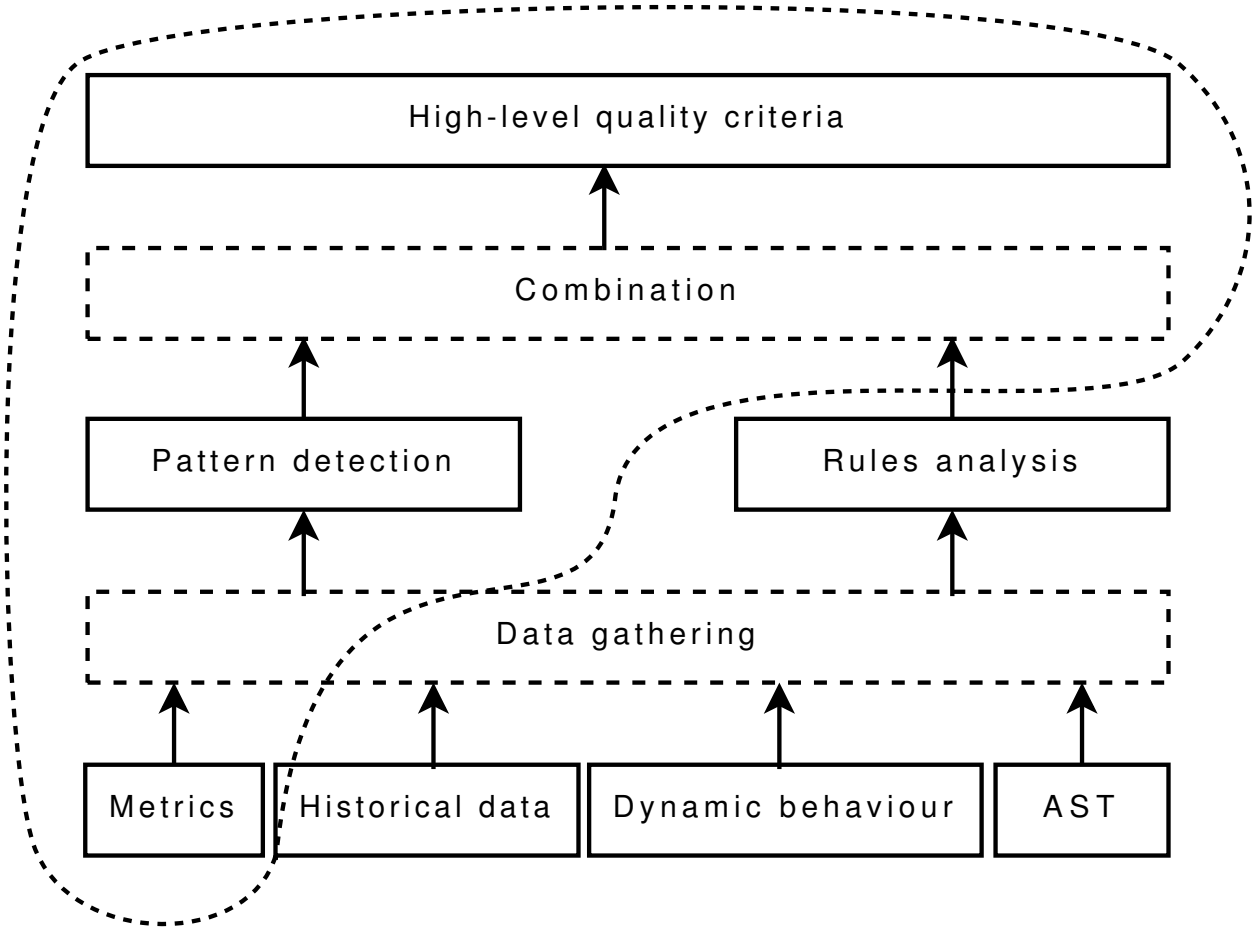


Figure 1. Hierarchical quality model

increase interpretability of the results. New patterns and existing strategies may be built with extended spectrum of data coming from new sources.

4.2.2. Metrics Quantization

As pointed out by Marinescu in [17], a simple vector of metrics values is not very useful, because there is no clear connection between measures and quality factors. In other words, such values require of proper interpretation. The method presented below provides a new interpretation mechanism for metrics, so that violations of rules can be detected and presented to the designer in intuitive way. In the context of the violated rules, we require an answer to the question: is the value of a metric unacceptable and, in consequence, measured characteristic has negative impact on quality? The simplest solution introduces a threshold: if a value of a metric exceeds threshold, then the measured attribute is considered to negatively impact the quality. The domain of the metric is divided into two intervals, which can be labelled as "negative impact" and "no impact". Thus, the labels provides interpretation for metrics values.

However, strict threshold values are inflexible, because values close to the threshold can be interpreted incorrectly in certain context. To provide a simple fix for that, the strict threshold value can be replaced with an additional interval representing the uncertainty. Values which falls into this interval should be analysed separately or supported by other data sources for correct classification.

Having considered these arguments, we can define three classes (intervals) of the attribute domain:

1. L – a value of a metric is unambiguously acceptable, and the measured attribute has no or negligible negative impact on quality,

2. M – a value of a metric is near to threshold; additional analysis is required or other data sources should be explored,
3. H – a value of a metric is unambiguously unacceptable, and the measured attribute has negative impact on quality.

We can formally define the labelling phase in following way:

1. E – a set of analysed entities, for example a class or a package,
2. M – a set of all metrics, suitable for the constructed model,
3. L – a set of all labels which identify classes of impact,
4. P – a set of all principles considered in the model,
5. m – a metric (e.g. CBO),
6. $m(e), e \in E$ – a value of metric m for entity e.

$$mli_{e,m} = \alpha_m(m(e)), e \in E, m \in M, mli_{e,m} \in L. \quad (1)$$

Function described by formula (1) maps a value of a metric m, measured for entity e, to a label mli[1]. As an effect, a numerical value delivered by a metric is replaced by a higher-lever label, which is already interpreted from the quality point of view.

The entire effort in the construction of this part of the model must be devoted to defining the α function. For the basic version of the model (with three classes) at least one threshold value with surrounding interval must be defined. The crucial step deals with identification of a threshold and a width of the interval.

The quantised metric – the labelled value – is only the very first and preliminary interpretation step. This information is valuable in larger context, thus labelled metrics should be utilised in compound patterns and strategies.

4.2.3. Entity-level Aggregation

Some of the characteristics and mechanisms, which constitute the basis for the rules of good design, are so complicated that there is a need for many supporting data sources, to capture all aspects and variations of those characteristics (e.g. coupling can be divided into import and export). Therefore, an aggregation function of a set of quantised metrics and other data sources has to be engaged, to answer the question: *Does a compound attribute, expressed by a set of input data, have a negative impact on quality?* Let be defined:

1. M_p – a set of metrics to express principle p, in other words, a set of metrics suitable for detection of violations of the principle,
2. $A_{e,p}$ – a set of all additional pieces of information, extracted from the other data sources (not metrics), for entity e and principle p,
3. $M_{e,p} = \{(m, mli_{e,m}) : e \in E, p \in P, mli_{e,m} \in L, \forall(m \in M_p) mli_{e,m} = \alpha_m(m(e))\}$ – a set of pairs: metric with assigned label; the label is assigned respectively to formula (1); the set is evaluated for all metrics referring to principle p and calculated for entity e.

$$pli_{e,p} = \beta_p(M_{e,p}, A_{e,p}), e \in E, p \in P, pli_{e,p} \in L. \quad (2)$$

Function defined by formula (2) aggregates a set of labelled metrics and additional information to label pli[2], which denotes impact of underlying characteristic on quality. Aggregation defined by formula (2) may be also realized as a classifier[3]. Assuming labels $l \in L$ denotes classes, the classifier built for specific principle p will assign a class l to an entity e. Meaning of the aggregated label or class can be generalised as follows: label $l \in L$ denotes strength of negative impact of an attribute upon quality.

Aggregation step requires careful interpretation of collected results, especially in the case of compound characteristics. To sum up above considerations:

1. well-known principles of software design are always based upon internal quality characteristic,
2. such characteristics can be decomposed into elements which can be later evaluated by data coming from diverse data sources. The collected results are useful for detection of violations of principles,

[1] Metric-level impact.
[2] Principle-level impact.
[3] For example using decision rules or trees.

3. aggregated results say nothing about the quality characteristic they are based on, but *provide information about the negative impact of a measured attribute on quality.*

Label evaluated by formula (2) denotes impact, but do not identify a violation of a principle. To define a violation, let be assumed:

1. VL_p – a set of labels, which are treated as a violation of principle p,
2. V_p – a symbol of a violation of rule p.

$$pli_{e,p} \in VL_p \Rightarrow V_p, e \in E, p \in P. \tag{3}$$

Definition If aggregated label *pli* for a characteristic supporting principle p, for analysed entity e, belongs to the set VL, then the entity is flawed by a violation of rule p.
This definition is captured by formula (3).

The detected violations can be scored and ranked just like Detection Strategies. As a consequence, presented method can be homogeneously in-lined with methods presented in Factor-Strategy model.

5. Example of Application

This section brings through a process of instantiation of a fragment of the hierarchical model. Scope of the example is narrowed to the elements which constitutes novelty of the model: rules analysis method with metrics quantization and aggregation. Instantiated model will be applied to exemplary entities.

5.1. Model Creation

5.1.1. Goals

The very first step of a model creation is the selection of quality characteristic to be evaluated. Following activities, like principles and metrics selection, are made in the context of the high-level quality goal. For the purpose of this example, readability (but analysability and understandability are closely related) of code and design is selected as a goal and high-level quality factor.

5.1.2. Principles

Coupling concept is considered to be a good predictor of quality. El-Emam in [10] provides evidence that high coupling makes programs hard to understand. Rule of low coupling, identified by Coad and Yourdon in [7] is selected as the design principle used as quality criterion in this example. Hence, let us define a set of principles $P = \{LowCoupling\}$.

5.1.3. Data Sources

For the purpose of coupling measurement, metrics *Ca* and *Ce*, defined by Robert Martin in [18], are used. The metrics count incoming (*Ce*) and outgoing (*Ca*) couplings separately, and will be applied at class level. Additional information, based on abstract syntax tree, is defined as a flag indicating whether an entity (a class in this case) is abstract. Let us assume:

1. $M = M_{LowCoupling} = \{Ca, Ce\}$ – a set of all metrics is actually the set of metrics for the design principle *LowCoupling*, because only one design principle is considered,
2. $A = \{\text{IsAbstract}\}$ – additional information from a non-metrics source.

5.1.4. Definition of Quantization and Aggregation

As described in [10] by [19], a human can cope with 7 ± 2 pieces of information at a time. We use this observation as a threshold for the above-selected coupling measures. For a quantization purpose, let us define:

1. $L = \{L, M, H\}$ – the basic set of labels,
2. $\alpha_{Ce}(Ce(e))$:

$$mli_{e,Ce} = \begin{cases} L, Ce(e) < 5 \\ M, Ce(e) \in [5, 9] \\ H, Ce(e) > 9 \end{cases} \tag{4}$$

3. $\alpha_{Ca}(Ca(e))$:

$$mli_{e,Ca} = \begin{cases} L, Ca(e) < 5 \\ M, Ca(e) \in [5, 9] \\ H, Ca(e) > 9 \end{cases} \tag{5}$$

The model is oriented toward detection of violations, so the simple max function will be used for aggregation, assuming that labels are ordered from the lowest value of L to highest H. Martin in [18] argues that classes should depend upon the most stable of them (eg. on abstract classes), so if a class is abstract then export coupling (Ca) is not taken into consideration. Aggregation function $\beta_{LowCoupling}(M_{e,LowCoupling}, A_{e,LowCoupling})$ is defined as follows:

$$pli_{e,LowCoupling} = \begin{cases} mli_{e,Ce}, & IsAbstract(e) \\ \max\{mli_{e,Ce}, mli_{e,Ca}\}, & \\ \quad \text{otherwise} & \end{cases} \tag{6}$$

Finally, let us define the violation:

1. $VL_{LowCoupling} = \{M, H\}$ – a set of labels indicating violations of *LowCoupling* rule; label M is also included to capture entities which probably violate the rule,
2. $V_{LowCoupling}$ – a symbol which denotes violation of *LowCoupling* rule,
3. $(pli_{e,LowCoupling} \in VL_{LowCoupling}) \Rightarrow V_{LowCoupling}$ – definition of *LowCoupling* rule violation.

5.2. Application

The model will be applied on sample data, taken from a student project, depicted in table 1. All classes are large (from 384 lines to 477 lines in a file) and probably flawed in many aspects. Results generated by the model are compared to results gathered in a survey, conducted among graduate software engineering students (students were asked to identify classes that are too large).

The quantized metrics and additional data for all entities:

1. $M_{DisplayManager,LowCoupling} = \{(Ce, H), (Ca, M)\}$
2. $M_{AmeChat,LowCoupling} = \{(Ce, H), (Ca, H)\}$
3. $M_{DrawableGroup,LowCoupling} = \{(Ce, L), (Ca, H)\}$
4. $A_{DisplayManager,LowCoupling} = \{IsAbstract = False\}$
5. $A_{AmeChat,LowCoupling} = \{IsAbstract = True\}$
6. $A_{DrawableGroup,LowCoupling} = \{IsAbstract = False\}$

Results of aggregation of quantized metrics:

1. $pli_{DisplayManager,LowCoupling} = \max\{H, M\} = H$
2. $pli_{AmeChat,LowCoupling} = mli_{AmeChat,Ce} = H$
3. $pli_{DrawableGroup,LowCoupling} = \max\{L, H\} = H$

Regarding the previous definitions of violations, all entities violate the principle of low coupling and negatively affect the high-level quality criterion.

5.2.1. Interpretation

The high-level quality goal – readability – is not evaluated because there are too few entities to get a relevant output. Let be assumed, the high-level factor indicates a problem in software. The very first step is to look for strategies and principles which support the factor, and choose only those with current negative consequences. The second step is to look for entities (suspects) which negatively impacts the factor in the context of chosen principle (or strategy). In this particular example there are only three classes and all of them are suspects due to violations of the principle.

Violation in *DisplayManager* results from the metric *Ce*, labelled with *H*, and *Ca* labelled with *M*. Considering *Ce* definition, *DisplayManager* suffers mainly from import coupling, and moderately from export coupling. Respondents classified *DisplayManager* as *Middle*

Table 1. Sample data

Class	*Ce*	*Ca*	$mli_{e,Ce}$	$mli_{e,Ca}$	*IsAbstract*
DisplayManager	13	8	H	M	False
AmeChat	14	35	H	H	True
DrawableGroup	4	14	L	H	False

Man and *Large Class*, and model results can indicate causes of these smells.

AmeChat is an abstract class, so it is obvious that it is used by many other classes. In consequence, only import coupling is considered, so the impact results from *Ce*, despite of high value of *Ca*. The vast majority of the respondents identified *Large Class* smell, which can be connected with high import coupling.

DrawableGroup uses desirable amount of classes, *Ce=L*, but is used in many other places. The majority of the respondents identified *Refused Bequest* in the class. This smell deals with inheritance, which is not considered in this model. Obtained results indicates other, coupling-related problems which probably cannot be named as a defined smell.

6. Summary

The proposed hierarchical model extends the Factor-Strategy model in three ways. It delivers more comprehensive and traceable information concerning detected potential anomalies to the designer, including the interpretation of metrics values, and also broadens the spectrum of analysed data sources to the non-metric ones. As the simple example suggests, these elements help in discovering new types of anomalies and also support the designer in evaluating the impact, scope and importance of the violation. It also delivers hierarchically structured data justifying the suspected flaws, and includes a uncertainty interval. Therefore, the model more resembles the human way of cognition.

Further directions of research include an experimental validation of the model, defining detection strategies utilizing data from heterogeneous data sources, and also embedding internal design characteristics into the model.

References

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[2] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[3] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25:1, 1999.

[4] L. C. Briand, W. L. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. Technical report, ISERN, 2000.

[5] L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22:68–86, 1994.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] P. Coad and E. Yourdon. *Object Oriented Design*. Prentice Hall, 1991.

[8] K. Dhambri, H. A. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *12th European Conference on Software Maintenance and Reengineering 2008*, pages 279–283, April 2008.

[9] F. B. e Abreu and R. Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th International Conference on Software Quality*, 1994.

[10] K. E. Emam. *Advances in Software Engineering*, chapter Object-Oriented Metrics: A Review of Theory and Practice, pages 23–50. 2002.

[11] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] R. Glass. On design. *Journal of Systems and Software*, 52(1):1–2, May 2000.

[14] T. L. Graves, A. F. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26:653–661, 2000.

[15] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. Using process history to predict software quality. *Computer*, 31:66–72, 1998.

[16] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty. A pattern-based framework for

software anomaly detection. *Software Quality Control*, 12(2):99–120, 2004.

[17] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[18] R. Martin. OO design quality metrics. An analysis of dependencies. *Report on Object Analysis and Design*, 2(3), 1995.

[19] G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, (63):81–97, 1956.

[20] D. Ratiu, S. Ducasse, T. Grba, and R. Marinescu. Using history information to improve design flaws detection, 2004.

[21] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001.

[22] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002.

[23] B. Walter and B. Pietrzak. Multi-criteria detection of bad smells in code with UTA method. In *Proceedings of XP 2005 conference*, pages 154–161, 2005.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Pattern-Based Software Architecture for Service-Oriented Software Systems

Claus Pahl*, Ronan Barrett*
*School of Computing, Dublin City University
Claus.Pahl@computing.dcu.ie, Ronan.Barrett@computing.dcu.ie

Abstract

Service-oriented architecture is a recent conceptual framework for service-oriented software platforms. Architectures are of great importance for the evolution of software systems. We present a modelling and transformation technique for service-centric distributed software systems. Architectural configurations, expressed through hierarchical architectural patterns, form the core of a specification and transformation technique. Patterns on different levels of abstraction form transformation invariants that structure and constrain the transformation process. We explore the role that patterns can play in architecture transformations in terms of functional properties, but also non-functional quality aspects.

1. Introduction

The development of distributed software systems based on service architectures is rapidly gaining momentum. *Service-oriented architecture* (SOA) is emerging as a new design paradigm and conceptual framework for distributed service-centric software systems, supported by platforms such as the *Web Services Framework* (WSF) [2]. Services are reusable software components that are explicitly described, published and provided at fixed locations. Due to the ubiquity of the Web, the WSF platform and SOA paradigm play a major role for software systems.

In service-centric distributed environments such as the Web services platform that allows services to be invoked using Internet protocols, a notion of workflow processes is central to capture service composition and interaction between services. We present techniques to support, firstly, modelling of services and service-oriented processes and, secondly, property-preserving transformations of service-oriented architectures. In contrast to a variety of architecture approaches that focus primarily on static, structural properties, we concentrate on dynamic dependencies in the form of interaction processes between services. Our solution is an approach to the *architectural transformation* of services, supporting the evolution of service-oriented architectures. Three aspects characterise our approach:

- Architecture modelling using hierarchical patterns. A *three-layered architecture model* addresses different levels of abstraction. Each layer is supported by a *pattern-based modelling approach* for service processes. A service-oriented architectural *configuration notation* that combines patterns and process behaviour in architectures forms the backbone. Patterns enhance reuse in SOA.
- Property-preserving architectural transformation. Based on the configuration notation as the abstract description language for source and target architectures, a *transformation technique* is developed. Patterns are considered as characteristics of a service architecture that are, due to the implied

reliability and maintainability, worth being preserved in transformations.
- Distribution and quality-of-service. We investigate the role of *distribution* for modelling and look at functional and non-functional service properties. The integration of *quality* aspects into modelling is important for the services platform, where providers and users are usually from different organisations.

We address the lack of behaviour and quality aspects in service-oriented architectural transformations. Our patterns capture essential behavioural service dependencies in the form of interaction process patterns and link these to quality properties. We utilise patterns to capture these properties and allow these properties to be preserved in transformations by identifying patterns as invariants. Formality is required to obtain unambiguous models of process-based service architectures and to complement modelling by analysis and reasoning facilities. Architectural change and integration require a technique for process-oriented property-preserving transformations.

We introduce our architecture model and transformation technique in Section 2. Pattern-based architecture modelling and specification, supported by the architecture configuration notation, is addressed in Section 3. Architectural transformations are defined in Section 4. Finally, we discuss related work and end with conclusions.

2. Architecture Model and Specification

Based on background definitions of service and software architecture, we now define the principles of our architecture model and the core notation.

2.1. Service-oriented Architecture

The objective of *software architecture* is the separation of computation and communication. Architectures are about *components* (i.e. loci of computation) and *connectors* (i.e. loci of communication). Various *architecture description languages* (ADL) and modelling techniques have been proposed [17]. An architectural model captures common concepts in architectural description: components provide computation, interfaces provide access and connectors provide connections between components. In service architecture, the main emphasis is on the composition of services to workflow processes and on the overall configuration of services and service processes. For instance [10], use *scenarios* – descriptions of interactions of a user with a system – to operationalise requirements and map these to a system architecture. We extend the notion of interaction and also consider system-internal interactions and allow interaction processes to be composite.

We focus on service architectures, i.e. service-oriented software architectures, here. A *service* is usually defined as a coherent set of operations provided at a certain location [2]. A service provider makes an abstract interface description available, which can be used by potential service users to locate and invoke this service. The Web Service platform provides description languages (WSDL) and invocation protocols (SOAP) for this purpose. Services are often used 'as is' in single request-response interactions. More recently, research has focused on the *composition of services* to *processes* [2]. Orchestration is the prevalent form of service composition. Existing services can be reused to form business or workflow processes. The *principle of architectural composition* that we look at here is *process assembly*.

2.2. An Architectural Configuration Notation

At the core of our architecture modelling and transformation technique is a conceptual architecture model. The objective of this conceptual architecture model is to capture the core layering and structuring principles of service-oriented architectures. The conceptual **service architecture model** (SAM), tailored towards the needs of service- and process-oriented platforms,

shall address the different abstraction levels and perspectives in service-oriented architectures:

- **Reference architectures** are high-level specifications representing common structures of architectures specific to a particular domain or platform.
- **Architectural design patterns** are medium-scale patterns – usually referred to as design patterns or architectural frameworks.
- **Workflow patterns** are process-oriented patterns that represent common data exchange-oriented workflow processes in an application domain.

Based on the architecture model, we define a notation for architectural specification – the **service-oriented architectural configuration** notation (SAC) – that has features of an abstract architectural description language (ADL). Two elements define our transformation technique: a *description notation* to capture architectural properties and *rules and techniques* for transformation.

Various formal approaches to the representation of processes have been suggested in the past, e.g. [6] using Petri nets. *Process calculi* such as the π-calculus [15, 13] are suitable frameworks for architectural configurations of service- and process-centric systems, i.e. support of modelling and transformation, due to their abstraction from service implementation and their focus on interaction processes. The π-calculus, a calculus for mobile processes, is particularly useful due to a similarity between mobility and evolution – both are about changes of a service in relation to its neighbourhood – which helps us to support architectural transformations. Our notation is defined in terms of the π-calculus [15], but we want to firstly provide a less mathematical syntax and, secondly, allow the addition of further combinators to express workflow and design patterns. A simulation notion captures property-preservation and permitted structure and behaviour variations during transformation.

Our notation consists of process activities, combinators and abstractions, which are summarised in Fig. 1. The basic element describing process activity is an **action**. Actions π are combined to **service process expressions**. Actions of a service are primitive processes divided into invocations and activations. **Invocations** **inv** x(y) by a client of a service via channel x connects to the remote service, passing y as a parameter. **Activations** receive **rcv** x(a) from a provider from other services and the dual reply **rep** x(b), with channel x and parameters a and b. Based on actions, process combinators are

Actions:

π	::=	**inv** $x(y)$	Invocation
		rcv $x(a)$	Activation – Receive
		rep $x(b)$	Activation – Reply

Processes – workflow combinators:

P	::=	π	Action
		$P_1; P2$	Sequential Composition
		par (P_1, P_2)	Parallel Composition
		repeat (P)	Iteration
		choice (P_1, P_2)	Exclusive Choice
		mchoice (P_1, P_2)	Multi-Choice

Processes – other constructs:

P	::=	**let** $x = \pi$ **in**	Variable
		0	Inaction

Abstraction:

$$A(a_1, \ldots, a_n) \quad = \quad P_A \quad \text{with } a_1, \ldots, a_n \text{ are free in } P_A$$

Figure 1. Syntactical Definition of the SAC Notation

basic forms of workflow patterns. **Sequences** are represented as P_1; P_2 – process P_1 is executed and the system transfers to P_2 where the next action is executed. **Exclusive choice** means that one P_i $(i = 1, \ldots, n)$ from **choice** $P_1, \ldots, P_n$ is chosen, **Multi-choice mchoice** $P_1, \ldots, P_n$ allows any number of the processes P_i $(i = 1, \ldots, n)$ to be chosen and executed in parallel. **Iteration repeat** P executes process P an arbitrary number of times. **Parallel composition par** $(P_1, \ldots, P_n)$ executes processes P_i concurrently. $A(a_1, \ldots, a_n) = P_A$ is a **process abstraction**, where P is a process expression and the a_i are free variables in P. A variable is introduced using **let** $x = \pi$ **in** P. **Inaction** is denoted by 0.

The semantics is defined in terms of the π-calculus [15], by mapping constructs directly to π-calculus constructs. The actions are defined in terms of send $\overline{x}\langle y \rangle$ (for invocation **inv** and reply **rep**) and receive $x(y)$ (for receive **rcv**) of the π-calculus. Combinators are defined through their π-calculus counterparts, except multichoice **mchoice** P_1, P_2, which is defined as **choice** (A, B, **par** (A, B)) – essentially a parallel composition of all elements of the powerset of the **mchoice** argument list. The abstraction is the π-calculus abstraction.

3. Pattern-Based Service Architecture Modelling

The architectural configuration notation SAC enables the modelling of pattern-based service architecture configurations.

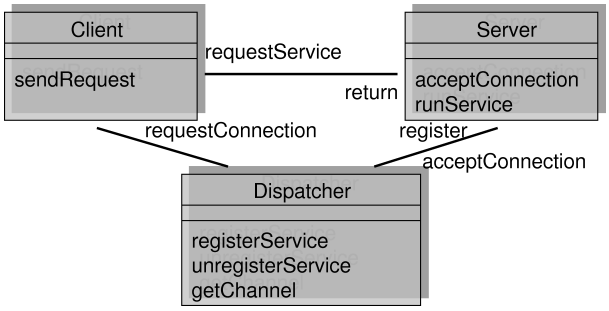
3.1. Patterns and Abstraction Levels

Architectural and *design patterns* are recurring solutions to software design problems [7]. Although originally proposed for object-oriented development, their applicability for service-based architectures has been demonstrated [18]. These patterns are about structure and interaction and provide reusable solutions to commonly encountered design problems. We use patterns at different levels of abstraction – reference architectures, architectural design patterns, and workflow patterns. We cover the three layers of the architecture model SAM. Workflow operators for service processes are directly integrated as operators. Architectural design patterns expressing service interaction patterns can be formulated as a number of concurrently executing processes. Reference architectures can be modelled at the level of abstractions.

Reference architectures, often emerge in an abstracted and standardised form from successful architectural assemblies. Reference architectures define accepted structures that help us to built maintainable and interoperable systems. Besides domain-specific architectures, which we will illustrate in the case study section, platform-specific reference architecture are important. Examples of classical Web-based architectures are *client-server* architectures or *three-tiered* architectures.

Design patterns are recognised as important building blocks in the development of software systems [7]. Their purpose is the identification of common structural and behavioural patterns. A rich set of design patterns has been described, which can be used to structure a software design at an intermediate level of abstraction. Usually, *architectural patterns* (such as model-view-controller) are distinguished from *design patterns* (such as factory, composite, or iterator) as the former are linked to component frameworks. We see both forms as intermediate-level constraints on a system architecture, i.e on services and on their interaction patterns.

Design patterns also play a role in the design of Web services architectures [18]. An example of an architectural design pattern in the Web services context is the *client-dispatcher-server* pattern [18]. The pattern architecture with its interactions is visualised in Fig. 2. The SAC notation adds behaviour specification to the static view of UML class diagrams. It is a textual description, similar to UML activity and interaction diagrams in purpose. We have used a UML class diagram to present the abstract service interface and the service connectivity. Pattern definitions such as *client-dispatcher-server* can




```
Client     = repeat (let requestServ = inv requestConnection()
                     in inv requestServ(resId))
Server     = inv registerServ(id);
             repeat (rcv acceptConnection(c); rcv requestServ(s);
                     rep requestServ(runService(s)))
Dispatcher = choice (
             choice (rcv registerServ(id), rcv unregisterServ(id)),
             repeat (rcv requestConnection();
               let c = getChannel()
               in  inv acceptConnection(c); rep requestConnection(c)))
```

Figure 2. *Pattern* – the Client-Dispatcher-Server Architectural Design Pattern

act as building blocks of complex architectures. Patterns are defined as process expressions and made available as process abstractions. These macro-style building blocks can also form a pattern repository.

Workflow patterns are small-scale process patterns [19] – often at the same level of abstraction as design patterns, but more focussed on data exchange. Workflow patterns relate to connector types that are used in the composition of services – we provide them as built-in operators. An example of a workflow pattern is the sequencing workflow pattern. Workflow patterns are small compositions of activities. Workflow patterns for Web services architectures are described in [20].

To identify workflow patterns in an architecture specification is important since often not all patterns are supported by the implementation language.

$$\mathbf{choice}(A, B, C, \mathbf{par}(A, B), \mathbf{par}(A, C), \mathbf{par}(B, C), \mathbf{par}(A, B, C))$$

is an equivalent workaround to the multichoice workflow, needed if the implementation language does not support the multichoice pattern **mchoice**(A, B, C) – which is the case with some WS-BPEL implementations [20].

3.2. Patterns and Quality

Patterns can influence a system's *quality characteristics* such as understandability or maintainability. For service-centric software systems specific properties arising from the often distributed and cross-organisational context are of central importance. The reliability of a system, the availability of services, and the individual service and overall system performance are often crucial.

- The qualtiy benefits of the client-dispatcher-server pattern are: composition is easy to *maintain*, as composition logic is contained at a single participant, the central dispatcher. *Low deployment overhead* as only the dispatcher manages the composition. Composition can consume participant services that are externally controlled. Web service technology enables the *reuse* of services.
- The main disadvantages are: a single point of failure at the dispatcher provides for *poor reliability/availability*. Communication bottlenecks at the dispatcher result in *restricted scalability*. Messages have considerable overhead for deserialisation and serialisation. A high number of often verbose messages

between dispatcher and clients/servers is sub-optimal and results in *poor performance*. All patterns have their advantages and disadvantages. Often, the qualities mutually affect each other negatively such as maintainability and performance. What is, however, important here is that the qualities associated to a given pattern are preserved during a transformation. The client-dispatcher-server pattern is typical for learning technology systems, for which maintainability and interoperability are central. Failure is not a highly critical problem and the number of users is predictable – which allows us to neglect two of the major disadvantages. Note, that these characteristics are associated to pattersn, but not part of our notation. For instance distribution is not part of our notation. We can use an annotation for the composition operators to indicate a distributed implementation if an extension is considered.

3.3. Case Study – Modelling Service Architectures

Our case study system is a learning environment called IDLE – the Interactive Database Learning Environment [14], which is based on object technology with a Web-based access interface. IDLE is a multimedia system that uses different mechanisms to provide access to learning content, e.g. Web server and a (synchronised) audio server. It is an interactive system that integrates components of a database development environment (a design editor, a programming interface, and an analysis tool) into a teaching and learning context. Learners can develop database applications, supported by shared storage and workspace.

IDLE has been developed since 1996 in several stages. The consequence of this growth is a system without a designed architecture – an architecture that is even not explicitly captured and documented. However, the existing architecture is service-oriented (although not fully Web Service-based) and, consequently, is a suitable starting point for transformations. Evolving Internet technologies and frequently changing software developers are only two of the contributors to difficult maintenance. Besides achieving maintainability, interoperability and componentisation were reasons to choose a fully service-based architecture as the target.

Architecture modelling starts with the proposed three-layered approach. In the context of our case study domain, the IEEE-defined *Learning Technology System Architecture* (LTSA) provides a domain-specific service-oriented reference architecture [9], visualised in the UML-style class diagram in Fig. 3. Six central components such as Delivery or Coach are identified. These components provide services, e.g. the Delivery component provides a Multimedia delivery service to the LearnerEntity. These services are usually related to processing multimedia data. We use the LTSA reference architecture as a starting point for the re-engineering of IDLE.

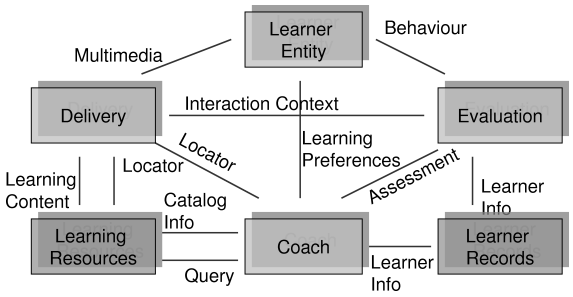


Figure 3. Overview of the LTSA *Reference Architecture*

In IDLE, a learner requests content from a resources server. The IDLE specification in SAC, Fig. 4, is based on the client-dispatcher-server design pattern, Fig. 2, with the learner (as client), a coach (as dispatcher), and the resources and delivery subsystem (as server). This specification captures a central behavioural property of IDLE, captured using the pattern. It is an extension of the pattern in terms of the IDLE application context that adds interaction with the Resources server to the Delivery component. Servers register their services with the dispatcher and clients request connection channels to servers in order to use the services. The learner is a client invoking services of the delivery (request a connection and an educational service). The coach is a broker and mediator that handles the service registration (from the

```
Learner  =  repeat (let requestEducServ = inv requestConnection()
                    in   inv requestEducServ(resId))
Delivery =  inv registerEducServ(id);
            repeat (rcv acceptConnection(c); rcv requestEducServ(s);
                    rep requestEducServ(run(s)); rcv locator(uri);
                    let learnResource = inv retrieveResource(uri)
                    in  rep multimedia(learnResource))
Coach    =  choice (
            choice (rcv registerEducServ(id), rcv unregisterEducServ(id)),
            repeat (rcv requestConnection();
              let c = getChannel()
              in  inv acceptConnection(c); rep requestConnection(c)))
```

Figure 4. *Specification* – Educational Service (EducServ) Registration and Provision in IDLE directly based on the Client-Dispatcher-Server Design Pattern

delivery) and forwards the delivery channel (provided by the delivery component) to the learner. Passing channel names over channels, as in the example, is typical for the notation's ability to model dynamic infrastructures. A learner uses the provided channel to access the delivery's educational service.

Another example of a design pattern is the *factory method pattern* – a creational pattern [7] that provides an interface for creating related objects without specifying their concrete classes. This pattern can be applied in IDLE for manipulating a variety of related persistent stores such as the learners records or adding/retrieving objects to/from a database such as a workspace feature.

Workflow patterns are the final architectural aspect. The multichoice operator denotes a process composition pattern. **mchoice**(Lecture, Tutorial, Lab) expresses that any selection of the IDLE services Lecture, Tutorial, and Lab can be used concurrently. We have realised the storage and workspace function, which could have been integrated into either learning resources or learner records, as a separate service. This IDLE feature can be specified as a complex service workflow process, see Fig. 5. The workspace service deals with incoming retrieval or storage requests.

Our service modelling notation needs a methodological context that covers modelling existing systems and transformations. Service-oriented architecture usually starts with the identification of services. Two cases can be distinguished:

- Some system components will exhibit service character – an SQL execution element, part of the IDLE lab resources and delivery subsystem, is an example.
- Some components could easily be wrapped up as services, if required. An example of this category is the IDLE storage and workspace feature.

Once all services have been identified, the connections and interactions between services have to be modelled. In our case study, the problem is re-engineering of a legacy system into a service-based system. The existing architecture – even though not adequately designed and documented – provides a starting point for service identification. The LTSA also determines the service-based modelling of IDLE due to the LTSA's SOA character. We have used a top-down approach to service identification as the first step of the transformation part.

The need to change, adapt and extend makes it clear that the original architecture cannot be fully preserved. An abstraction mechanism – in the form of patterns – answers the need to focus on essential, but not all architectural properties that should be preserved. Patterns not only identify common functional structures; they also have typical quality attributes associated with them. A central difficulty arises: how to identify suitable patterns. The collection of frequent patterns is often domain-specific, as our investigation indicates. Examples of frequently occurring design patterns in IDLE, other learning technology systems, and also the LTSA include the client-dispatcher-server

```
WorkSpace = choice (
                repeat (rcv retrieve(resId); inv provide(res)),
                repeat (rcv store(resId, res)))
```

Figure 5. *Specification* – Specification of the IDLE Storage and Workspace Service

pattern, but also the factory, proxy, observer, composite, and serialiser patterns [7]. Other, less frequent patterns include the iterator and the strategy pattern. These common patterns could result in a domain-specific formulation of patterns and a repository of domain-specific patterns, which would help software architects in identifying invariants of the transformation.

4. Transformation

Software architecture addresses more than the high-level system design. Software change resulting from maintenance requirements and integration problems is equally important. We focus on architecture transformations as a central software change technique. A number of reasons might require transformations:

- Interoperability can be a transformation objective.
- A reference architecture might need to be adopted.
- Changes in interface and interaction of services need to be addressed.

Architectures are often transformed if implementation restrictions have to be dealt with. An objective of architecture transformation is to implement changes, but also to preserve properties. Existing service connectivity and interaction is often worth being preserved, i.e. act as invariants of the transformation. Our patterns express processes at different levels of abstraction. Preserving patterns is desirable since patterns represent architectural configurations that are easy to understand and implement and describe structures that are often easy to maintain and reliable.

While the idea of preserving patterns at all architecture layers is therefore obvious, a verifiable transformation technique is needed. A generic constructive mapping rule is at the centre of our transformation technique. A notion of simulation captures the notions of equivalence and refinement of services and service processes.

A prerequisite for transformations is the explicit architecture specification of an existing system. A complete specification is not necessary; accuracy and level of preservation of the transformation, however, depend on the degree of detail and number of patterns identified. In IDLE, we have for instance analysed an inadequately documented system to extract structures and patterns.

4.1. Simulation and Transformation Rules

Our transformation technique is based on a notion of simulation and on simulation-based transformation rules. It has to address the needs of the three pattern-based architecture layers and the focus on patterns as transformation invariants. Each of the three architecture models might create its own requirements:

- Reference architectures. Each service abstraction is mapped to a service abstraction in the new architecture. The transformation objective determines whether the service process definition has to be changed. The transformation is subject to invariants, i.e. pattern preservation.
- Architectural design patterns. Often, interaction processes need to be changed to accommodate new or modified service functionality. Ideally, newly emerging patterns that a service participates in will simulate the original patterns.
- Workflow patterns. Workflow pattern transformations can often be handled automatically in architecture implementations.

Property preservation is the goal of our architecture transformations. A simulation notion shall capture service process pattern preservation in the transformation technique. A *simu-*

lation definition, adopted from the π-calculus, satisfies the pattern preservation requirement for the processes that we envisage:

> Process Q **simulates** process P if there exists a binary relation $\mathcal{S}$ over the set of processes such that if whenever $P\mathcal{S}Q$ and $P \xrightarrow{m} P'$ then there exists Q' such that $Q \xrightarrow{n} Q'$ and $P'\mathcal{S}Q'$ for service processes n and m.

This definition expresses when process Q based on service expression n preserves, or simulates, the behaviour of process P based on service expression m. The services n and m can here be unrelated, as this definition is about observable behaviour only.

In order to automate transformation support based on this definition, a constructive theorem supporting simulation is needed. This theorem is the basis of a transformation rule which allows the verification of preservation and the automation of transformation. In [12], we have developed a constructive simulation test based on the construction of transition graphs for SAC process expressions.

Since usually not the entire specified behaviour should be preserved, we have introduced the notion of patterns to capture common behavioural aspects that need to be preserved. Patterns at different levels of abstraction identify reliable and maintainable interaction patterns between services. Central to our transformation technique is a **transformation rule**, which associates patterns and simulation:

> Given an architecture specification S in SAC, create an architecture specification S' as follows. For each abstraction A in S (apply this rule recursively from top to bottom), **map** A **to** A' where A' is another abstraction such that for any pattern P that A participates in, A' *simulates* P' with $P' = P[A/A']$, i.e. A' substitutes A and P is replaced by P' to cater for renaming of abstractions.

This produces pattern-preserving target architectures, if no further modification are made. We, however, argue that further modifications of the initial architecture in terms of additional or modified functionality are typical for transformations in evaluation and integration contexts. In this case, the invariant pattern preservation needs to be demonstrated. *Pattern-preserving transformation rules* can aid here. These are based on standard simulation relationships discussed in the process algebra literature [15], such as:

- $A; B$ simulates A: only transitions of B are added that do not affect A.
- **repeat**(A) simulates A: a single repetition corresponds to A.
- **choice**$(A; B)$ simulates A: the selection of A corresponds to A.
- **par**$(A; B)$ simulates A: A is always executed in the parallel composition.

From this constructive rule set, pattern-preserving transformations that even include structural and behavioural changes can be formulated.

The determination of an invariant, here the pattern P, is a common, but often non-trivial problem, which can be alleviated through domain-specific patterns.

4.2. Case Study – Pattern-Preserving Transformations

We demonstrate the adoption of the LTSA reference architecture on the highest level of abstraction for the IDLE system. The transformation aim is interoperability of IDLE services and components with other LTSA-specified components. This interoperability objective, however, can have an impact on all levels of abstraction. Other learning technology standards, for instance, prescribe interfaces for learning technology objects, which would have to be reflected in service interfaces here.

The starting point for the transformation is the architecture specification of an existing system – in our case IDLE in its original form. IDLE on the highest level of abstraction is a parallel composition of composite processes

$$\begin{aligned}\text{IDLE} = \textbf{par}(\ &\text{Learner}, \text{Delivery}, \text{StudentModel},\\ &\text{PedagogyModel}, \text{Workspace},\\ &\text{Evaluation}, \dots\ )\end{aligned}$$

where each top-level service is an abstraction of a process expression based on other, more

basic services. Some of these are already similar to LTSA components – we have indicated this fact by using the similar names. Other existing IDLE components such as StudentModel and PedagogyModel have no direct counterpart in the LTSA, but can be abstracted by e.g. the Coach. Several different combinations of individual services can form patterns; these might actually overlap.

The first transformation step is to describe IDLE's architectural characteristics – ideally in LTSA terminology to simplify the transformation, see Fig. 3. The client-server-dispatcher pattern, see Fig. 2, is not identical to the structure that can be found in the IDLE system, see Fig. 4, since interactions with the resources server are added. The pattern itself as an identifiable pattern is nonetheless worth preserving and is, thus, one of the invariants. In our case, the *client-dispatcher-server* pattern **par** (Client, Dispatcher, Server) is therefore *simulated* by the composite IDLE process **par** (LearnerEntity, Coach, Delivery), resulting from the composition of learner, coach, and resources and delivery subsystems of the IDLE reformulation in LTSA terminology. This property is in our case easy to verify, since the IDLE specification in Fig. 4 describes only the service requests and connections that establish functionality defined in the pattern.

LTSA is a high-level system specification, to which we add functionality in IDLE in the form of new services not covered by LTSA. Architectural changes are necessary due to the workspace service integration into IDLE. The explicit storage and workspace service, see Fig. 5, requires the services LearnerEntity and Delivery to be modified in their interaction behaviour. Again, the pattern shall be the invariant of the transformation, but some refinements – constrained by the simulation definition – need to be made to accommodate the added service within the system.

Workflow patterns to be preserved can be identified due to their implementation as operators in the notation. The specification of the IDLE educational service system based on the client-dispatcher-server architectural design pattern in Fig. 4 based on Fig. 2 is defined in terms of workflow patterns. The Learner is based on a sequence of activities. The Coach is based on choice in the first part, and a concurrent split and merge in the second part. These are candidates for invariants.

The reconstructed IDLE architecture is the transformation basis. The integration of specifications of the identified existing or created services forms the transformed architecture. The transformation task is to transform IDLE into LTSA-IDLE – an architectural variant of IDLE with LTSA-conform service interfaces and interaction processes. In the transformation, we need to consider the source, the invariant, the target construction, and the preservation proof.

- **Source.** The starting point of the transformation is the original IDLE specification. Since in our case a full specification did not exist, we analysed the system and extracted its current structural, behavioural and quality properties based on existing documentation and system tests. The high-level architecture was given earlier and some detailed excerpts are presented in Figs. 4 and 5.
- **Invariant.** The invariant is determined by patterns on different levels of abstraction. The LTSA determines the high-level architecture. We focus here on the client-dispatcher-server pattern as the architectural pattern invariant. The identification of patterns as invariants is a crucial and difficult step that depends on the expertise of the software architect – domain-specific patterns with common behaviour or qualities provide a starting point for invariant identification. The central pattern that we have identified and chosen to be an invariant captures the interactions between three of the central components of IDLE, i.e. learner, coach and delivery. It is one of the patterns that we found frequently in learning technology systems, and that we considered suitable to capture common interaction behaviour between central system components.
- **Target Construction.** The LTSA-based architecture specification of some IDLE services – which is the transformation result –

```
LearnerEntity = repeat (let requestEducServ = inv requestConnection()
                        in inv requestEducServ(resId);
                           let preferencesInfo = inv getPreferences();
                               learnResource = inv multimedia()
                           in inv setPreferences(alter(preferencesInfo)))
Delivery      =  inv registerEducServ(id);
                 repeat (rcv acceptConnection(c); rcv requestEducServ(s);
                         rep requestEducServ(run(s)); rcv locator(uri);
                         let learnResource = inv retrieveResource(uri)
                         in rep multimedia(learnResource))
Coach'        = choice (
                 choice (rcv registerEducServ(id), rcv unregisterEducServ(id)),
                 repeat (rcv requestConnection();
                   let c = getChannel()
                   in  inv acceptConnection(c); rep requestConnection(c);
                       repeat (
                         choice (
                           rcv getPreferences(); rep getPreferences(prefInfo),
                           rcv setPreferences(preferencesInfo),
                           rcv getLearnerInfo(id); rep getLearnerInfo(info),
                           let uri = inv locator(resource) in 0))))
LearningRes   =  rcv retrieveResource(uri); rep retrieveResource(retrieve(uri))
LearnerRec    =  rcv getLearnerInfo(id); rep getLearnerInfo(info(id))
```

Figure 6. *Transformation* – Resulting Adaptive Delivery in IDLE Architecture (selected components and services) based on the LTSA

can be found in Fig. 6. It is constructed based on our transformation rule as follows.

- At the reference architecture level, IDLE is mapped to LTSA-IDLE where the merger of StudentModel and PedagogyModel simulates the Coach. This requires a reformulation of the IDLE process (parallel composition of composite processes, e.g. Delivery) as LTSA-IDLE by renaming abstractions and introducing Coach as a new element on the highest level.
- At the architectural design pattern level, the composition is changed at the subcomponent level. Coach is defined to reflect the merger of the two model components as a parallel composition of StudentModel and PedagogyModel.
- **Simulation and Preservation.** The invariants – LTSA and client-dispatcher-server – are two patterns that have to be simulated by the new architecture. We have adapted our terminology to LTSA. For instance, Learner becomes LearnerEntity. Renaming does not affect the simulation property. The two components StudentModel and PedagogyModel are merged into Coach, i.e. the model components were abstracted by a single Coach interface, which results in the LTSA pattern being simulated. In this case, Coach is only introduced as an abstraction for behaviour that already existed in the source system. Simulation is therefore also guaranteed. The new Coach' service handles the interaction with the learner and pedagogy model components. The original Coach specification from Fig. 4 has been extended to reflect this fact, which is presented in Fig. 6. The structural and behavioural properties of the client-dispatcher-server pattern P := **par**(Client, Dispatcher, Server) are still intact, i.e. the pattern is preserved according to the transformation with pattern P and the original Coach adapted to Coach'. The three pattern components are still present and the externally visible interaction behaviour is the same[1].

[1] The formal proof is based on a constructive simulation test developed in [12], which is beyond the scope of this paper.

The specification in Fig. 6 describes a more complete range of interactions than the initial focus of Fig. 4 on educational service request and connection establishment. Fig. 6 adds the adaptive delivery of resources. After updating preferences by interacting with the coach, the learner entity requests and receives learning resources via a multimedia channel from the delivery service. The learning resources service retrieves the actual content for the delivery service, which in turn delivers it to the learner entity. Adding functionality or significantly modifying the original architecture is common in evolution and integration situations. This is also the primary motivation for introducing invariants that are abstractions of the original architecture. The original architecture can due to these modifications in practice rarely be fully preserved – only well-chosen abstractions can be suitable invariants.

Verifying the preservation of the client-dispatcher-server invariant in the resulting architecture is a non-trival task. We can demonstrate for each affected service, i.e. LearnerEntity, Coach' and Delivery, that each simulates the original component:

- LearnerEntity simulates the Client through the first process elements (the repeat expression with the first two invocations) in the sequence of four subprocess expressions. In general, $\mathbf{repeat}(A; B)$ simulates $\mathbf{repeat}(A)$ because for each state transition in A there is a corresponding one in $A; B$.
- Delivery (which is unchanged) simulates the Server since, similar to the first case, only basic activities such as receive-reply interactions and invocations with the Resources component are added within the repeat loop.
- Coach' simulates the Dispatcher as the Dispatcher functionality becomes the outer process structure of the Coach to which preferences and learner initialisation and the location retrieval aspects are added. $A; B$ simulates A because transitions in A are also part of $A; B$.

Constructive rules are important in discharging the simulation proof obligation.

In our method, *design patterns* that can be identified in an existing system such as the original IDLE, should be *invariants of the architectural transformation*. This method can be supported by transformation tools. The architect provides the source system model and identifies preservable patterns from the model patterns and, if necessary, renamings and non-standard transformations. More involvement from the software architect is required if in the context of the transformation process, architectural features are also changed or extended. In this case, which is actually the standard situation in application integration and software migration, a fully automated approach is not feasible and the software architect needs to apply the provided constructive transformation rules to guarantee pattern preservation.

5. Related Work

Some ADLs are similar to SAC in terms of their focus on processes. Darwin [11] is a π-calculus based ADL. Darwin focuses on component-oriented development approach, addressing behaviour and interfaces. Restrictions based on the declarative nature of Darwin make it rather unsuitable for the design of service-based architectures, where flexibility and change demands such as both binding and unbinding on demand are required features. Wright [1] is an ADL based on CSP as the process calculus. Wright supports compatibility and deadlock checks through formalised specifications, based on explicit connector types. This is an aspect that we have neglected here, but that could enable further analysis techniques, if we introduced typed channels. In [5], the formal foundations of a notion of behaviour conformance are explored, based on the π-calculus bisimilarity relation. We chose the π-calculus as our basis, since it caters for mobility, and, consequently, allows us to address transformation in the context of architecture evolution. Mobility allows us to deal with changes in the interaction infrastructure. The client-dispatcher-server pattern is an example where a new channel is dynamically

formed. Architecture transformation also means controlled changes of architectural structures.

Patterns have recently been discussed in the context of Web service architectures [18, 19, 20, 4]. In [19, 20], collections of workflow patterns are compiled. We have based our catalog on these collections. The client-dispatcher-server pattern is also discussed in [18]. Other patterns that we have mentioned mainly originate from [7]. Grønmo et al. [16] consider the modelling and building of compositions from existing Web services using model-driven development. The authors consider two modelling aspects, service (interface and operations) and workflow models (control and data flow concerns). These efforts embed patterns into a methodological framework, similar to our objectives. Our consideration of distribution as a further dimension in service patterns, however, goes beyond those approaches.

A recent software architecture approach for service-based systems is model-driven development (MDD). MDD emphasises the importance of modelling and transformations. The latter are, in contrast to our framework, part of the modelling process between modelling levels of abstraction. Our framework addresses the transformation of architecture specifications, for instance to support software change and evolution. While MDD is vertically oriented, i.e. mapping from abstract domain models to more concrete platform models, we follow a more horizontal transformation approach on the level of architectures. We have focused on hierarchical pattern-based process modelling and architectural configuration – two aspects that can complement and extend MDD by providing higher levels of abstraction and architectural transformation. The formality of our approach satisfies the automation requirements of model-driven development and even adds reasoning support.

6. Conclusions

A new architectural design paradigm such as service-oriented architecture (SOA) requires adequate methodological support for design, maintenance, and evolution. While an underlying deployment platform exists in the form of Web Services, an engineering methodology and techniques are still largely missing. We have presented a layered architecture model that captures behavioural aspects and associates quality of architectural structures at different levels of abstraction through patterns. A modelling notation allows interaction behaviour in architectures and architectural configurations to be captured and distribution and quality characteristics to be associated. Interaction behaviour and composite processes are essential aspects for the development and maintenance of distributed service-based systems.

Our emphasis here was on the applicability of the method by demonstrating the usefulness for a service-based learning technology system. We have investigated the role that hierarchically organised patterns, supported by the architecture model and the transformation technique, can play for service-oriented architecture. Patterns that capture interaction behaviour between services are ideally suited for the service context with its focus on processes. Process patterns provide an abstraction mechanism that captures relevant invariants for architectural transformation.

- Patterns as abstractions greatly improve the possibility to reuse and evolve architectural designs. As architectural abstractions, they capture important behaviour and quality invariants.
- Pattern-based modelling has implications for functional and quality characteristics of a service-centric software system. Pattern-based transformation focuses on functional properties, but also preserves the quality characteristics.

The novelty of our architecture transformation technique is to use patterns to capture behaviour and quality invariants in a layered architectural modelling approach to service-based architecture evolution and change.

We have applied the presented techniques in the ongoing design, maintenance and evolution of the IDLE system. It is an extensive system with a range of interactive, distributed features, characterised by complex a information archi-

tecture, that has been developed by more than 20 people and maintained for more than ten years – which indicates the scalability of the transformation technique. The technique was described in its principles and illustrated using the case study. Our tool implementation for distribution pattern architecture demonstrates the positive effect of pattern-based transformations on architectures in terms of quality. However, the pragmatics of modelling with formal notations need to be addressed further. While in the case study, architects were familiar with the notation, a closer integration with UML activity diagrams is envisaged to improve acceptance and usability.

A critical aspect of the approach is the reliance on the quality of the architectural description of the original system and the adequacy of the identified patterns – particularly obvious is migration and legacy integration projects. Transformations depend on the detail of the input architecture and the patterns that define the transformation invariant. The extraction of a system's architecture and the correct identification of intended patterns for undocumented systems is a difficult aspect that, although essential for the success, has been addressed only through the idea of domain-specific patterns here. Re-engineering and migration approaches for the architectural level can provide further solutions here.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):249, 1997.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: concepts, architectures and applications*. Springer Verlag, 2004.

[3] R. Barrett, L. M. Patcas, C. Pahl, and J. Murphy. Model driven distribution pattern design for dynamic web service compositions. In *Proceedings of the 6th international conference on Web engineering*, page 136, 2006.

[4] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[5] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2):105–138, 2001.

[6] R. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *International journal of cooperative information systems*, 13(4):337–368, 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[8] D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In T. Cant, editor, *Proceedings of the eleventh Australian workshop on Safety critical systems and software*, page 17, 2007.

[9] IEEE P1484.1/D8. Draft standard for learning technology – learning technology systems architecture LTSA, 2001.

[10] R. Kazman, S. J. Carrière, and S. G. Woods. Toward a discipline of scenario-based architectural engineering. *Annals of software engineering*, 9(1–4):5–33, 2000.

[11] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Software Engineering—ESEC'95*, pages 137–153.

[12] C. Pahl. An ontology for software component matching. *Fundamental Approaches to Software Engineering*, pages 6–21.

[13] C. Pahl. A Pi-calculus based framework for the composition and replacement of components. In *SAVCBS 2001 Proceedings*, page 97, 2001.

[14] C. Pahl, R. Barrett, and C. Kenny. Supporting active database learning and training through interactive multimedia. *ACM SIGCSE Bulletin*, 36(3):31, 2004.

[15] D. Sangiorgi and D. Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[16] D. Skogan, R. Grønmo, and I. Solheim. Web service composition in UML. In *Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings*, pages 47–57, 2004.

[17] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. Software architecture: Foundations, theory, and practice. 2009.

[18] N. Y. Topaloglu and R. Capilla. Modeling the variability of web services from a pattern point of view. *Web Services*, pages 128–138.

[19] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[20] M. Vasko and S. Dustdar. An analysis of web services workflow patterns in collaxa. *Web Services*, pages 1–14.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

The Evolution of Complexity in Apple Darwin: A Common Coupling Point of View

Liguo Yu*
*Computer Science and Informatics, Indiana University South Bend
ligyu@iusb.edu

Abstract

Common coupling increases the interdependencies between software modules. It should be avoided if possible. In previous work, we presented two types of categorization of common coupling, one is for single-kernel-based software, one is for multi-kernel-based-software. In this paper, we analyze the relationships between these two types of categorization and apply them to study the evolution of the complexity of Apple Darwin. The same conclusion about Darwin's evolution is drawn based on the two types of categorization of common coupling: From version XNU-517 to version XNU-792, Darwin has restructured to reduce the number of difficulty-inducing high category (level) global variables in order to reduce the system complexity. However, due to the definition-use dependencies, the complexity of Darwin induced by global variables has increased from version XNU-517 to version XNU-792.

1. Introduction

Coupling measures the degree of dependencies between two software modules [9, 6, 5]. Strong coupling indicates a high degree of dependencies while loose coupling indicates a low degree of dependencies. High degree of dependency makes the software modules difficult to maintain and reuse. For example, to identify the origin of a fault, the best practice is to separate modules and test each of them individually. Loose coupling with low degree of dependencies can make the fault isolation process easier, while strong coupling with high degree of dependencies will make this process tedious and time/effort consuming. Consider reuse, it is easier to reuse a module that has loose coupling and is weakly dependent on others than a module that has strong coupling and is tightly dependent on others. Therefore, from the viewpoint of maintenance and reuse, a good software system should have low coupling between modules.

The software couplings can be divided as data coupling (simple data are passed as parameters in a function call), stamp coupling (data structures are passed as parameters in a function call), external coupling (two modules access the same file/database), and common coupling (two modules access the same global variable), in which, common coupling is considered to be a strong form of coupling. That is, common coupling induces high degree of dependencies between software modules and accordingly makes software modules difficult to understand, maintain, and reuse [8, 7].

Software evolution is inevitable. On the one hand, software needs to continually satisfy customers' functional requirements and non-function requirements. On the other hand, software needs to promptly adapt to the changes of hardware and system environments. Therefore, with the evolution of a software system, new features and new modules to support new hardware, are continually added to the source code. Both the size of the product and the com-

plexity of the product are expected to increase as new versions are developed and released. At the same time, to make the system align with its original quality design, the code structure needs to be monitored and examined frequently, and restructuring should be taken as needed to reduce the system complexity in order to achieve high maintainability and reusability. One way to reduce the system complexity is to replace existing strong couplings or new strong couplings introduced in the evolution process with loose couplings.

In this paper, we use common coupling as a measure of the system complexity and study how it changes with the evolution of a software system. The study is performed on Apple Darwin, an open-source operating system. The objective of this study is to understand the changing patterns of software complexity under the dual effects of size increasing and code restructuring in the evolution process.

The remainder of the paper is organized as follows: Section 2 describes kernel-based software. Section 3 reviews the categorizations of common coupling. Section 4 presents the study of the evolution of Darwin. The conclusions and limitations appear in Section 5.

2. Kernel-Based Software

Many software products, such as operating systems and database systems, are called *kernel-based software* [2]. That is, the software system consists of architecture and/or platform independent kernel modules, together with specific architecture and/or platform dependent nonkernel modules [8, 1]. Software product line is another example of kernel-based system, in which, the core assets are considered as kernel modules, and custom assets are considered as nonkernel modules. Figure 1 depicts the production of kernel-based software: Each implementation/installation of kernel-based software involves the use of all kernel modules and optional nonkernel modules.

In previous work [17], we identified two types of kernel-based software, single-kernel-based

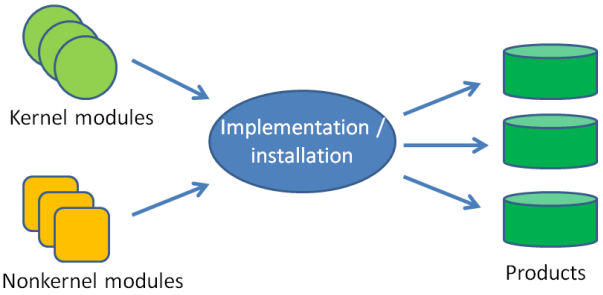


Figure 1. Depiction of the production of kernel-based software

software and multi-kernel-based software. In a kernel-based software system, if all the kernel modules are included in one component and the rest nonkernel modules are included in another component, we call it *single-kernel-based software*. Figure 2 shows the structure of a single-kernel-based software system, in which circles represent kernel modules, squares represent nonkernel modules, and rectangles represent components. In other words, in single-kernel-based systems, kernel modules and nonkernel modules are clearly separated into two components, kernel component and nonkernel component [17]. Examples of single-kernel-based systems are Linux and BSDs.

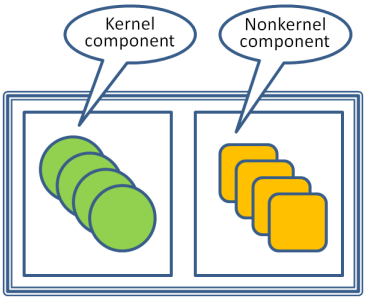


Figure 2. Depiction of single-kernel-based software

In a kernel-based software system, if the kernel modules are included in more than one component, we refer to that system as *multi-kernel-based software*. Figure 3 shows a multi-kernel-based software system, in which, kernel modules (represented with circles) and nonkernek modules (represented with squares) coexist in multiple components. These components that consist of both kernel modules and nonkernel modules are called *kernel-based components* and are represented with triple line rect-

angles. We use the term *outer component* to refer to the software component external to the kernel-based components. The outer component consists of no kernel modules and is represented with a dashed triple line rectangle. Examples of multi-kernel-based software are Apple Darwin and TrustedBSD SED Darwin [10].

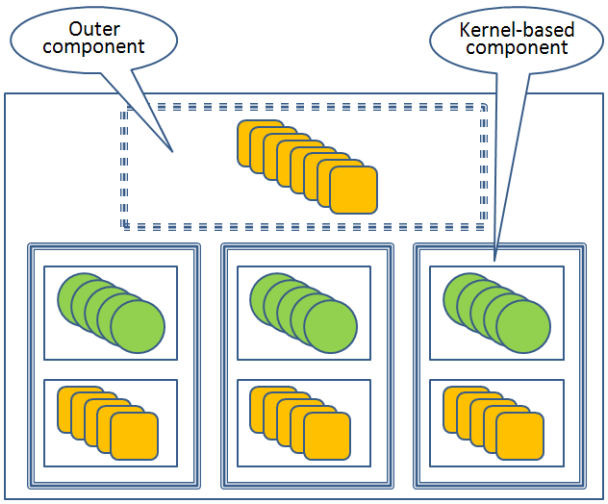


Figure 3. Depiction of multiple-kernel-based software

3. Categorizations of Common Coupling in Kernel-Based Software

In previous work, we presented two types of categorization of common coupling, one is within a single-kernel-based software system [15, 13, 16], and one is within a multiple-kernel-based software system [17]. These categorizations provide two approaches to measuring the maintenance effort and reuse effort of a kernel-based software system [14, 12]. These categorizations are reviewed here to provide the background knowledge about this research. Both of these categorizations are related with the definition-use analysis of global variables [15, 17].

3.1. Definition-use Analysis

The occurrence of a variable in a source code statement is related with one of the two tasks: reading the value of the variable or writing a value to the variable. Writing a value to a variable is called *definition* of a variable. The most common form of variable definition is an assignment statement, such as $\mathbf{x = 10}$. Reading the value of a variable is called *use* of a variable. The use of a variable is a statement that utilizes the value of the variable, such as **print(x)**. From the declaration of a variable to the destruction of that variable, each time the variable is invoked, it is either assigned a new value (a definition) or its present value is used (a use).

Common coupling induces dependencies between software modules through the definition-use of a global variable. For example, if module **M1** defines a global variable and module **M2** uses that global variable, we say that module **M2** is dependent on **M1** via common coupling. This dependency induced by definition-use of a global variable has effects on both software maintenance and reuse. Considering maintenance, if changes are made to module **M1**, attentions must be given to module **M2** to examine the effects of such changes and if necessary, corresponding changes should be made on **M2**. For reuse, if we want to reuse module **M2**, we must consider either reusing **M1** together with **M2** (because **M2** is dependent on **M1**), or modifying **M2** to remove its dependence on **M1**.

3.2. Categorization of Common Coupling in Single-Kernel-Based Software

In previous work, we divided global variables in single-kernel-based software into 5 categories, as shown below [15].

- Category 1: A global variable is defined in one or more kernel modules but not used in any kernel modules.
- Category 2: A global variable is defined in one kernel module and is used in one or more kernel modules.
- Category 3: A global variable is defined in more than one kernel module, and is used in one or more kernel modules.
- Category 4: A global variable is defined in one or more nonkernel modules and is used in one or more kernel modules.
- Category 5: A global variable is defined in one or more nonkernel modules and is

defined and used in one or more kernel modules.

In these five categories, high categories (Categories 4 and 5) global variables are considered worst for kernel maintenance and reuse, because they induce dependencies of kernel modules on nonkernel modules; Categories 2 and 3 global variables induce dependencies locally within the kernel and are accordingly considered better than Categories 4 and 5; Category-1 global variables do not affect kernel dependencies and are considered better than all others. For more discussions about these five-category global variables, the readers are referred to [15].

3.3. Categorization of Common Coupling in Multiple-Kernel-Based Software

In previous work [17], we divided global variables in multiple-kernel-based software into 6 levels. They are listed below.

- Level 0: A global variable is defined in kernel modules but not used in kernel modules.
- Level 1: A global variable is defined and used within the same kernel module but not defined in any other modules.
- Level 2: A global variable is used in kernel modules and is defined in nonkernel modules of the same kernel-based component.
- Level 3: A global variable is used in kernel modules of one kernel-based component and is defined in nonkernel modules of an outer component.
- Level 4: A global variable is used in kernel modules of one kernel-based component and is defined in nonkernel modules of another kernel-based component.
- Level 5: A global variable is used in kernel modules of one kernel-based component and is defined in kernel modules of another kernel-based component.

In these levels, a Level-0 global variable cannot affect the dependencies of kernel modules, because there is no use in kernel modules. Therefore, the presence of a Level-0 global variable will not cause difficulties in the maintenance and reuse of kernel modules. The definition and use of a Level-1 global variable are all within one kernel module and accordingly, this definition does not affect the dependency of this kernel module as well as other kernel modules.

In contrast, a kernel module that uses a Level-2 global variable depends on the nonkernel modules that define the global variable. This dependency is within the same kernel-based component and it does not affect other kernel-based components. High level (Levels 3 to 5) global variables might affect kernel maintenance and reuse. A Level-3 global variable induces dependencies of kernel modules on outer modules. Level-4 and Level-5 global variables are worst. They induce dependencies of kernel modules on modules of other kernel-based components. For more discussions about these six-level global variables, the readers are referred to [17].

3.4. Relations between the Two Types of Categorizations

In multiple-kernel-based systems, there are more than one kernel-based components. However, in some cases, we are interested in only one specific kernel-based component and consider the kernel modules within this component as kernels and all the rest modules (within or outside of this component) as nonkernels. Therefore, we can also consider this multi-kernel-based software system as a single-kernel-based software system. For example, in Figure 3, if we only consider the kernel modules within the first kernel-based component as kernels, the resulting system is single-kernel based, as shown in Figure 4.

Accordingly, the global variables categorized into six levels in multi-kernel-based software systems can be recategorized and mapped to the five categories in single-kernel-based software systems. The mapping and the relationships are shown in Table 1. Therefore, a global variable in multi-kernel-based software could be categorized using two schemes. In the remainder of this paper, we call the 5-category categorization *Categorization 1* and the 6-level categorization *Categorization 2*.

To study the effects of definition-use of a global variable on kernel dependencies, we utilize the following terminologies.

- **Dependency-inducing definition**: a definition of a global variable that can induce the dependency of kernel modules on other modules.
- **Non-dependency-inducing definition**: a definition of a global variable that cannot induce the dependency of kernel modules on other modules.
- **Safe dependency-inducing definition**: a definition of a global variable that induces the dependency of kernel modules on other kernel modules.
- **Unsafe dependency-inducing definition**: a definition of a global variable that induces the dependency of kernel modules on other nonkernel modules.

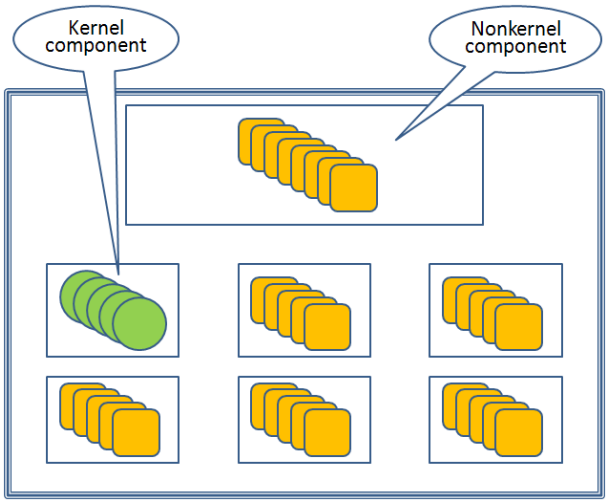


Figure 4. The single-kernel point of view of the system shown in Figure 3

Table 1. The mapping of 6-level categorization to 5-category categorization of global variables in multi-kernel based software

Level number (6-level categorization)	Category number (5-category categorization)
0	1
1	2, 3
2, 3, 4, 5	4, 5

In other words, if a definition of a global variable induces the dependency of kernel modules on other modules (either kernel or nonkernel), it is a dependency-inducing-definition. Otherwise, the definition is called a non-dependency-inducing definition. The definitions of global variables in Categories 2 to 5 (Levels 1 to 5) are dependency-inducing definitions. Definitions of global variables in Category 1 (Level 0) induce no dependencies of kernel modules and are accordingly non-dependency-inducing.

There are two types of dependency-inducing definitions, safe dependency-inducing definition and unsafe dependency-inducing definition. Safe dependency-inducing definitions are related with Categories 2, 3 and 5 (or Levels 1 to 5) global variables and these definitions occur in kernel modules. Unsafe dependency-inducing definitions are related with Categories 4 and 5 (or Levels 2 to 5) global variables and these definitions occur in nonkernel modules. Table 2 and Table 3 classify the definitions into different types. For example, a definition of Category-1 global variable in a nonkernel module is a non-dependency-inducing definition; a definition of a Level-2 global variable in a kernel module is a safe dependency-inducing definition; a definition of a Level-3 global variable in a nonkernel module is a unsafe dependency-inducing definition. The symbol "—" indicates there is no such definitions of a global variable in the corresponding category (level).

We remark here that the terminologies of safe/unsafe dependency-inducing definitions presented in this paper are different from the terminologies of safe/unsafe definitions cited in [15]. The unsafe definitions cited in [15] are actually the dependency-inducing definitions presented here and the safe definitions cited in [15] map to the non dependency-inducing definitions presented here.

4. The Evolution of Darwin

4.1. Overview

Darwin is Apple's open-source operating system for Macintosh computers. Figure 5 shows the architecture of Apple Darwin. It conceptually consists of three components: One kernel-based component (denoted as **osfmk**) that is reused (with modifications) from Mach [4], another kernel-based component (denoted as **bsd**) that

Table 2. The classification of definitions in single-kernel based software

Category	Definitions in kernel modules	Definitions in nonkernel modules
1	Non-dependency-inducing	Non-dependency-inducing
2	Safe dependency-inducing	–
3	Safe dependency-inducing	–
4	–	Unsafe dependency-inducing
5	Safe dependency-inducing	Unsafe dependency-inducing

Table 3. The classification of definitions in multiple-kernel based software

Level	Definitions in kernel modules	Definitions in nonkernel modules
0	Non-dependency-inducing	Non-dependency-inducing
1	Safe dependency-inducing	–
2	Safe dependency-inducing	Unsafe dependency-inducing
3	Safe dependency-inducing	Unsafe dependency-inducing
4	Safe dependency-inducing	Unsafe dependency-inducing
5	Safe dependency-inducing	Unsafe dependency-inducing

is reused (with modifications) from FreeBSD [3, 11], and the third component (denoted as outer) is a new written regular component. The structure of Darwin shown in Figure 5 indicates that it is a dual-kernel-based system. Darwin is written in C/C++, in the remainder of this paper, we use the term module to refer to a source code file written in C or C++ (**.c** file, **.cpp** file, or **.h** file).

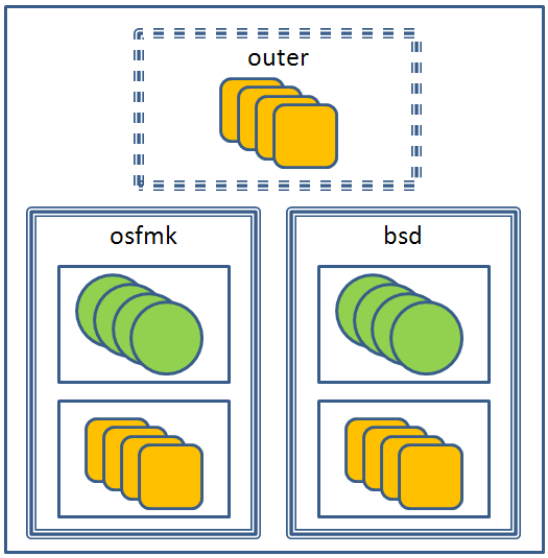


Figure 5. Architecture of Apple Darwin, a dual-kernel-based software system

To study the changes of kernel dependencies of Darwin with the evolution of the system complexity, we compared common coupling in two versions of Darwin, XNU-517 and XNU-792, which were released in November 2003 and April 2005, respectively. Figure 6 illustrates the evolution of Darwin from XNU-517 to XNU-792. It shows that the total number of modules increased about 5 percent and the total size measured in KLOC (Thousand Lines of Code) increased about 11 percent.

4.2. The Evolution of Complexity Induced by Common Coupling

In order to study common coupling in Darwin kernels, for each kernel module, we determined all the global variables and characterized them using two different schemes described in Section 3, *Categorization 1* and *Categorization 2*. Figure 7 and Figure 8 illustrate the evolution of the global variables in Darwin from the viewpoint of single-kernel-based software and multiple-kernel-based software respectively.

In both categorizations, we can see that the total number of global variables increased from version XNU-517 to version XNU-792. However, if we look at the most unfavorite global variables (Categories 4 and 5 in *Categorization 1* and Levels 2, 3, 4, and 5 in *Categorization 2*), the number decreased from 17 to 14 and from 23 to 19 for **osfmk** kernel and **bsd** kernel respectively. Therefore, we can conclude that, from version XNU-517 to version XNU-792, Darwin has restructured to reduce the number of high-level

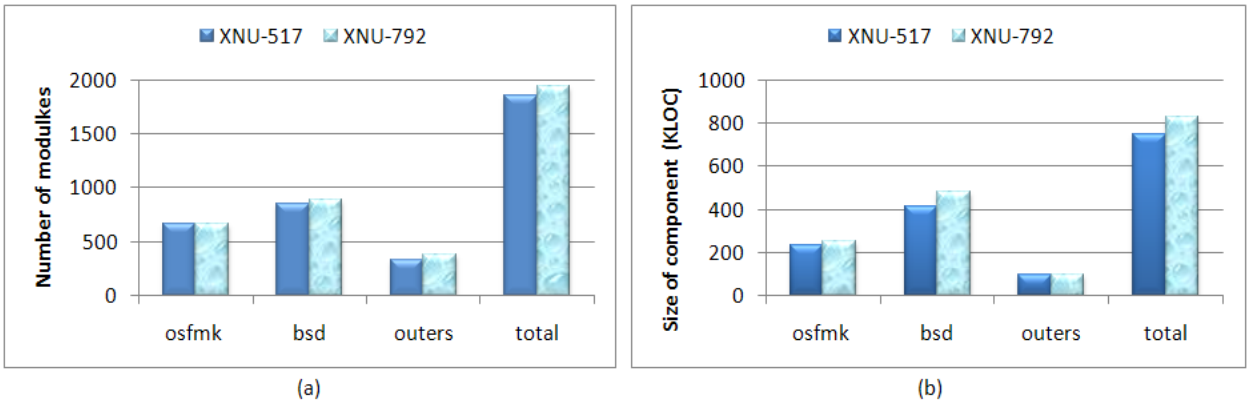


Figure 6. The evolution of Darwin from XNU-517 to XNU-791: (a) the number of modules; and (b) the size (KLOC) of the system

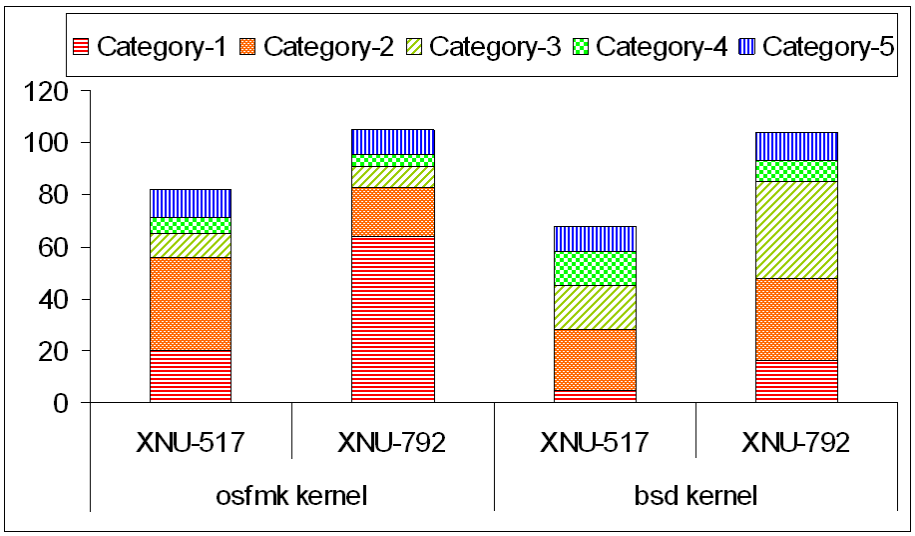


Figure 7. The number of global variables in Darwin kernels – *Categorization 1*

global variables, which are potential obstacles to kernel maintenance and reuse.

However, as we discussed before, the complexity induced by common coupling is related with the definition-use of global variables. To understand the evolution of the complexity in detail, we need to study the evolution of definition-use of global variables in Darwin from version XNU-517 to version XNU-792.

As described in Section 3, only the dependency-inducing definitions can affect kernel dependencies, we therefore studied the evolution of dependency-inducing definitions of global variables in **osfmk** kernel and **bsd** kernel. The results are shown in Table 4 and Table 5. The definitions are classified using the *Categorization 1* scheme, which applies to single-kernel-based software. It is worth noting that Category-1 global variables have non-dependency-inducing definitions and are accordingly not included in Table 4 and Table 5. Fig. 9 summarizes **osfmk** kernel (Table 4) and **bsd** kernel (Table 5) and shows the overall evolution of dependency-inducing definitions in Darwin. It can be seen that from version XNU-517 to version XNU-792, both the number of safe dependency-inducing definitions and the number of unsafe dependency-inducing definitions increased.

Table 6 shows the detail about the evolution of unsafe dependency-inducing definitions in Apple Darwin from the viewpoint

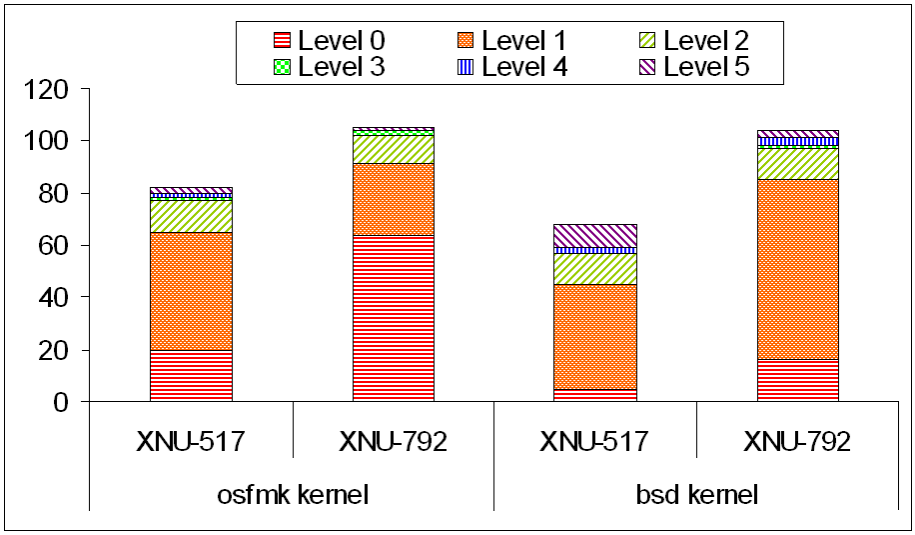


Figure 8. The number of global variables in Darwin kernels – *Categorization 2*

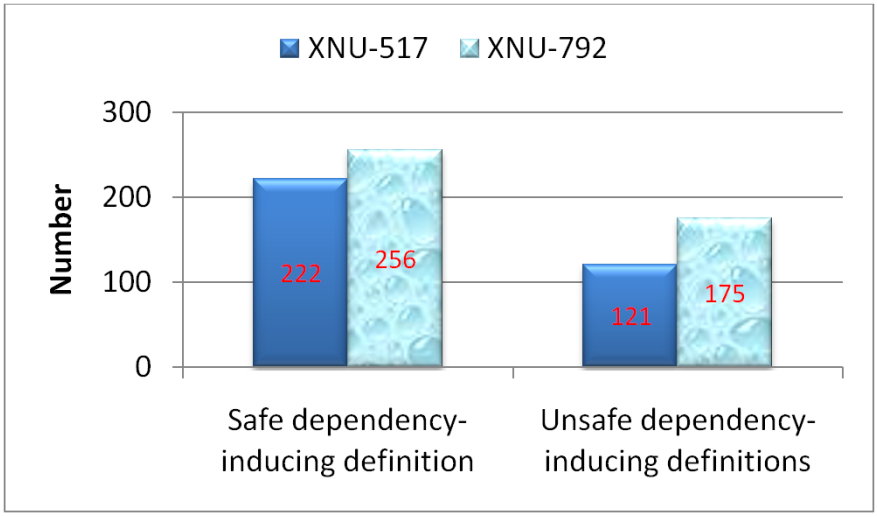


Figure 9. The evolution of the dependency-inducing definitions in Darwin

of multi-kernel-based system and the global variables are classified using *Categorization 2* scheme. It is worth noting that Level-0 and Level-1 global variables have no unsafe dependency-inducing definitions and are according not listed in Table 6.

As discussed in Section 3.3, high level (Level-4 and Level-5) global variables can bring more difficulties for kernel maintenance and reuse than low level (such as Level 2 and Level 3) global variables. Because the number of high level (Level-4 and Level-5) global variables is reduced from version XNU-517 to version XNU-792 (Figure 8), the number of unsafe dependency-inducing definitions induced by high level (Level-4 and Level-5) global variables decreased (Table 6). However, the number of unsafe dependency-inducing definitions induced by low level (Level 2 and Level 3) global variables increased tremendously (Table 6). It can be seen in Table 6, overall the number of unsafe dependency-inducing definitions increased from 121 in version XNU-517 to 175 in version XNU-792, matching the evolution of unsafe dependency-inducing definitions shown in Figure 9, which is derived from Table 4 and Table 5. Therefore, using two different categorization schemes, the same result is obtained: the complexity of Ap-

Table 4. The evolution of dependency-inducing definitions for **osfmk** kernel – *Categorization 1.*

Category number	Number of definitions			
	Definitions in kernel (Safe dependency-inducing)		Definitions in nonkernel (Unsafe dependency-inducing)	
	XNU-517	XNU-792	XNU-517	XNU-792
2	59	19	–	–
3	28	21	–	–
4	–	–	25	15
5	27	23	33	74
Sum	**114**	**63**	**58**	**89**

Table 5. The evolution of dependency-inducing definitions for **bsd** kernel – *Categorization 1.*

Category number	Number of definitions			
	Definitions in kernel (Safe dependency-inducing)		Definitions in nonkernel (Unsafe dependency-inducing)	
	XNU-517	XNU-792	XNU-517	XNU-792
2	34	32	–	–
3	52	134	–	–
4	–	–	48	15
5	22	27	15	71
Sum	**108**	**193**	**63**	**86**

Table 6. The evolution of unsafe dependency-inducing definitions in Darwin kernel – *Categorization 2*

Level number	**osfmk** kernel		**bsd** kernel	
	XNU-517	XNU-792	XNU-517	XNU-792
2	53	77	22	18
3	2	11	0	44
4	2	0	17	12
5	1	1	24	12
Sum	**58**	**89**	**63**	**86**

ple Darwin induced by global variables (common coupling) increased from version XNU-517 to version XNU-792.

4.3. Discussions

Software evolution is inevitable, because new features need to be frequently added and new hardware and platforms need to be continually supported. This is demonstrated in the evolution of Apple Darwin: From version XNU-517 to version XNU-792, both the size of kernel and the size of the entire system increased.

As new modules are added to the system, new dependencies need to be created between these new modules and existing modules, which will increase the complexity of the system. The maintenance activity performed on existing modules might also alter its original quality and increase the complexity of the system. Therefore, an evolving software system needs to be restructured regularly to retain its high quality design. This is also demonstrated in the evolution of Apple Darwin, in which, we found, from version XNU-517 to version XNU-792, Darwin has restructured through reducing the number of high category (level) global variables. This restructuring effort decreases the effect of the complexity increasing and dependency increasing due to the growth of the kernel size and the product size.

However, the definition-use analysis of the evolution of kernel dependencies shows that the number of dependency-inducing definitions, especially the number of the unsafe dependency-inducing definitions, increased from version XNU-517 to version XNU-792. Therefore,

the overall complexity of the system in the viewpoint of common coupling increased despite the effort of restructuring in reducing the number of unfavorite high category (level) global variables.

With the growth of the size of Darwin, both the module complexity and the module dependency are expected to continually increase. To reduce the effects of common coupling on kernel maintenance and kernel reuse, we suggest that major restructuring should be taken on Apple Darwin to reduce the number of dependency-inducing definitions, especially the number of unsafe dependency-inducing definitions.

5. Conclusions and Limitations

In this paper, we studied the evolution of the complexity of Apple Darwin from version XNU-517 to version XNU-792. We applied two schemes of categorization of common coupling, in which Apple Darwin is considered as both a single-kernel-based software system and a multi-kernel-based software system. Analysis of the two categorizations gives the same result. Specifically, the study found that from version XNU-517 to version XNU-792, the complexity of Apple Darwin increased. To reduced this increase of complexity, restructuring is necessary and it has been taken. Although the number of high category (level) unfavorite global variables is reduced in this restructuring process, the number of unsafe-dependency-inducing definitions increases. Therefore, the complexity of Darwin increased from the viewpoint of common coupling in spite of the effort of restructuring. Suggestions are that major restructurings should be taken to reduce or remove the unsafe dependency-inducing definitions in order to reduce the system complexity.

There are several limitations to this research. One limitation is that this research only focus on one type of coupling: common coupling; other types of component dependencies are not considered. The result could be improved if the evolution of more types of couplings are studied. Another limit is that only two versions of Apple Darwin are studied in this research. If more versions of Apple Darwin together with more versions of other operating systems are studied using the technique proposed in this paper, the result could be more interesting and convincing.

References

[1] P. B. Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 4(4):238–241, 1970.

[2] T. Härden. New approaches to object processing in engineering databases. In *Proceedings of International Workshop on Object-Oriented Database Systems*, pages 217–217, September 1986.

[3] Kernelthread. What is Mac OS X. http://www.kernelthread.com/mac/osx/, 2005.

[4] Mach. Mach 3.0 sources. http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/sources/, undated.

[5] J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *Journal of Systems and Software*, 20(3):295–808, 1993.

[6] M. Page-Jones. *The Practical Guide to Structured Systems Design.* Yourdon Press, New York, 1980.

[7] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt. Maintainability of the Linux kernel. *IEE Proceedings–Software*, 149(1):18–23, 2002.

[8] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt. Quality impacts of clandestine common coupling. *Software Quality Journal*, 11(3):211–218, 2003.

[9] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(13):115–139, 1974.

[10] TrustedBSD. http://www.trustedbsd.org/sedarwin.html, 2008.

[11] J. West. How open is open enough? modeling proprietary and open source platform strategies. *Research Policy*, 32(7):1259–1285, 2003.

[12] L. Yu. Common coupling as a measure of reuse effort in kernel-based software with case studies on the creation of MkLinux and Darwin. *Journal of the Brazilian Computer Society*, 14(1):45–55, 2008.

[13] L. Yu and S. Ramaswamy. Categorization of common coupling in kernel-based software. In *Proceedings of the 43rd ACM Southeast Conference*, volume 2, pages 207–210, March 2005.

[14] L. Yu, S. R. Schach, and K. Chen. Common coupling as a measure of reuse effort in kernel-based software. In *Proceedings of 19th International Conference on Software Engineering and Knowledge Engineering*, pages 39–44, July 2007.

[15] L. Yu, S. R. Schach, K. Chen, and J. Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Transactions on Software Engineering*, 30(10):694–706, 2004.

[16] L. Yu, S. R. Schach, K. Chen, J. Offutt, and G. Heller. Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. *Journal of Systems and Software*, 79(6):807–815, 2006.

[17] L. Yu, S. R. Schach, K. Chen, and S. Ramaswamy. Coupling measurement in multi-kernel-based software with its application to Darwin. *The International Journal of Intelligent Control and Systems*, 13(2):109–118, 2008.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Integration of Application Business Logic and Business Rules with DSL and AOP

Bogumiła Hnatkowska*, Krzysztof Kasprzyk*
*Faculty of Computer Science and Management, Institute of Informatics, Wrocław University of Technology
bogumila.hnatkowska@pwr.wroc.pl, krzysiek.kasprzyk@gmail.com

Abstract

Business processes and business rules are implemented in almost all enterprise systems. Approaches used today to their implementation are very sensitive to changes. In the paper authors propose to separate business logic layer from business rule layer by introducing an integration layer. The connections between both parts are expressed in a dedicated domain specific language (DSL). The definitions in DSL are further translated into working source code. The proof-of-concept implementation of the integration layer was done in the aspect oriented language (AOP) – AspectJ. The AOP was selected because it fits well to encapsulate scattered and tangled source code implementing the connections between business logic and business rules with the source code implementing core business logic.

1. Introduction

Software systems of enterprise class usually support business processes and business rules existing in a given domain. Because both (business processes and business rules) are often subject of change, they should be defined within a software system in such a way that is easy to maintain. Approaches used today to business rules implementation are very sensitive to changes, i.e. each modification of: (a) business rule set (b) when (within a business process) to fire specific business rule (c) which business rules to fire – can result in the necessity of application source code modification. Additionally, the source code implementing the connections between business logic and business rules is often scattered and tangled with the source code implementing core business logic. That allows to treat the problem of integration between business logic layer and business rules (considered as a separate layer) as a cross-cutting concern. A mechanism usually used for separation of cross-cutting concerns within software systems is Aspect Oriented Programing (AOP) [8], and one of the most popular AOP programming languages is AspectJ [8].

According to [10], business rules should be separated from business processes, however they apply across processes and procedures. What more, business rules should be expressed declaratively, and they should be executed directly, for example in a rules engine. In the paper authors describe an architecture of a software system, satisfying all above mention demands. The main element of the architecture is an integration layer that lies between business rules repository and business logic layer. The layer is implemented in AspectJ. Unfortunately, aspect-oriented languages are rather difficult, so the source code of intermediate layer is complex and hard to understand. Therefore there is a need for more abstract language (rather declarative one) which can be used for describing how

to integrate business logic with business rules. In this paper authors present a domain specific language (DSL) serving that purpose. Models written in the DSL are automatically translated to AspectJ source code. The DSL editor with syntactic checks as well as transformations were implemented in the oAW framework [9].

The structure of the paper is as follows. In chapter 2. main features of integration layer are presented. In chapter 3. the DSL syntax shortly is described. Short but complete examples are shown in chapter 4. Chapter 5. presents related works while chapter 6. contains some concluding remarks.

2. Features of Integration Layer

Business model defines basic notions from a given domain, the relationships between the notions and the way in which they are constrained. Business rules constitutes an important part of a business model. *A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business* [5]. There are many types of business rules, for example von Halle distinguishes [3]:

- terms – nouns which meaning is commonly accepted,
- facts – statements defining relationships among terms,
- rules – declarations of a policy or a condition that must be satisfied.

Rules are defined upon terms, and facts, and they are further divided into constraints, action enablers, inferences, and computations. Terms and facts usually are expressed directly in the source code of application. If they are changed also the source code is modified. Rules can be implemented either directly in the application source code or outside it. Using today approaches to rules realizations try to separate them into some components (modules, etc.) to minimize the influence of their changes on the other parts of application. The advantages of rules modularization are as follows:

- Rules are directly expressed;
- Rules can be shared between different business processes;
- It is easier to define who and when can modify rules;
- Rules can be maintained (update, create, delete) not only by programmers but also by business experts.

A typical solution to rules modularization employs business rule engines or business rule management systems like JBoss Rules [7], JRules [5], or Jess [4]. However, even in such a case, source code responsible for communication with the engine is scattered and tangled with application source code responsible for business logic. Additionally, every time you decide to use (or not to use) a rule in a given place, you need to modify the application business source code. To eliminate above mentioned problem we have decided to introduce separate layer in the application architecture, between business logic layer and rules representation – see Fig. 1. The main aim of this layer is to isolate the business logic layer from rules. So, this should prevent the business logic layer from being influenced by rules evolving or changing.

The desired features of integration layer are presented below:

- Support for invocations of all rule kinds;
- Definition when to invoke what rules;
- Passing parameters and context dependent information to rules.

The are two kinds of activation events that can result in rules invocation:

- method invocation event,
- attribute change event.

Business rules should be validated depending on the context that is changing dynamically. So, integration layer should allow to specify a dynamic context in which an activation event will result in business rule(s) firing. Authors have identified different time relationships between an activation event and rule(s) invocation – see Fig. 2. An activation event can cause rules invocation if it happened:

- during execution of specific business method,
- when specific business method is not executed,

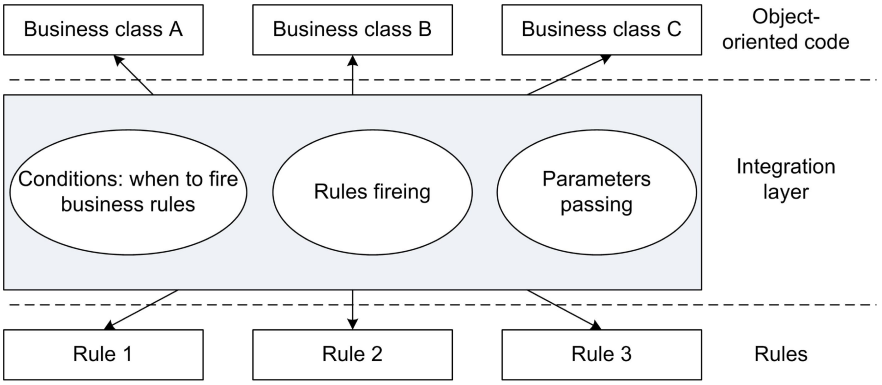


Figure 1. Integration layer in application architecture

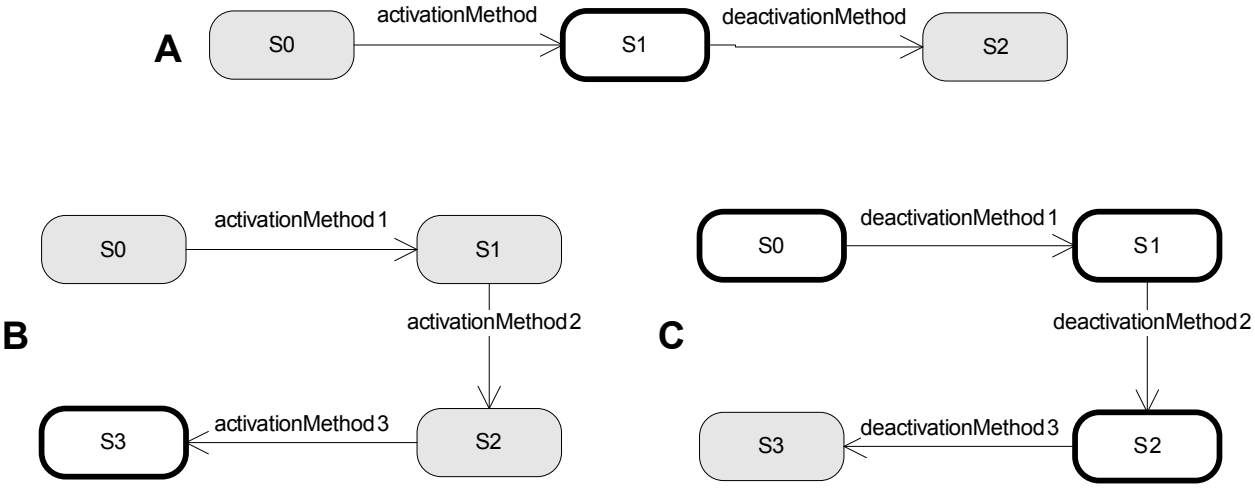


Figure 2. Possible scenarios of defining the moments of business rule invocation (gray state – the rule can not be fired, white state – the rule can be fired)

- after execution of activation method but before invocation of deactivation method (case A in Fig. 2; business rule can be fired in $S1$ state),
- after execution of a sequence of activation methods (case B in Fig. 2, where the sequence consists of three methods; business rule can be fired in $S3$ state),
- before execution of a sequence of deactivation methods (case C in Fig. 2, where the sequence consists of three methods; business rule can be fired in $S0$, $S1$ or $S2$ states).

The presented cases (A, B, C in Fig. 2) can be put together to define more complicated context.

3. DSL definition

To hide the complexity of integration layer a textual domain specific language was defined. It allows to specify how to integrate business logic with business rules in a declarative way. The full meta-model of the DSL consists of 29 classes and 42 relationships (17 generalizations, 25 associations). The concrete syntax of the language was defined in the form supported by the oAW framework. Models written in DSL are validated against a set of rules expressed in Check language which is a part of the Xtext framework. Transformation between models and As-

pectJ source code was implemented in Xpand template language. The general structure of text files written in DSL is presented below:

```
Package declaration
[Import section]
[Global object declaration]
Event definitions
Business logic to business rules link definitions
```

The presentation of the DSL is constrained to mandatory elements.

3.1. Package Declaration

Package declaration has the same form as in java program. It specifies where (to what package) generate AspectJ code.

3.2. Event Definition

Business rules are fired in strictly defined moments during program execution. As it was mention above there are two kinds of activation events: method invocation event, and attribute change event. Definition of an event activated by method invocation has a form presented below:

```
event <event name> isExecutionOf method
  <method name> in <type name>
  [withParams ( <parameter list> )]
  [returns <type name>]
end
```

The definition contains the unique name for the event and the signature of the method (optionally types of parameters and type of returned value).

As activation event is also responsible for preparing the context to business rules evaluation. By using `asFact` keyword some data associated with method execution are passed to the rules engine:

- The reference to an object realizing the method (`in <type name> asFact <object name>`);
- The value returned by the method (`returns <type name> asFact <object name>`);
- The references to objects passed as parameters (`withparams(<type name_1>, ..., <type name_k>) asFact <object name_1>,..., <object name_k>`).

The data will be used further in link definitions (see chapter 3.3).

Definition of an attribute change event has the form presented below:

```
event <event name> isUpdateOf field
  <attribute name> in <type name>
end
```

It defines the unique name for the event, the localization (class) and the name of the attribute. Similarly to the method activation event there is a possibility to exhibit some data:

- The new value of the attribute (`newValue <type name> asFact <object name>`);
- The object which attribute is modified (`in <type name> asFact <object name>`).

3.3. Business Logic to Business Rules Link Definition

Business logic to business rules links are the main elements of DSL. They are built upon events and values exhibited by them. Definition of the link determines what business rule(s) when to fire, and optionally the data necessary for business rules evaluation, context of execution etc.:

```
link <link name>
  [configuredBy <path to configuration file>]
  fires <rule names> <when clause> <event name>
  [requires <object name_1>,...,<object name_k>]
  [active <context definition>]
end
```

The most important part of the definition is `fires` clause. It is a regular expression defining the names of business rules that should be fired in a reaction to a specific event. The `when` clause specifies exactly when to run business rules. There are three possibilities to choose from:

1. `before` (rules are called before event activation);
2. `after` (rules are called after event activation);
3. `insteadOf` (rules are called instead of event activation).

The `requires` clause is used for passing necessary data identified by names to a rule engine. The order of object names is important because it determines the order of events that

are responsible for preparing the objects. The `active` clause defines the context (additional constraints) in which the activation event (defined in `fires` clause) results in business rules invocation. There are many possibilities for context definition, below are presented two of them:
- `while <event name>` – activation event must occur within flow of method defined by an event,
- `except <event name>` – activation event must occur outside flow of method defined by an event.

4. Examples

4.1. Example 1

Let consider a simple business process (called *Order Processing*) that aims at processing an order of a given customer. The process consists of four basic operations performed in a sequence:
1. order validation,
2. order's total cost calculation,
3. order's shipping cost calculation,
4. sending an order for further realization.

If an order is not validated (result of operation 1), status of the order is set to rejected and the whole business process is stopped; otherwise status of the order is set to accepted, and the process is continued. The business process is constrained with the following set of business rules:
- Rule 1: "Gold" customer pays 40% less for shipping.
- Rule 2: "Platinum" customer pays nothing for shipping.

The data model and the business layer model (limited to the considered example) of the application supporting *Order Processing* business process is presented in Fig. 3. An order contains a collection of order items, each of which has a price defined. An order is placed by a customer. A customer belongs to one of customer categories (regular, gold, platinum). The main class realizing the business logic is *OrderProcessingService* with *processOrder* operation. The operation implements all four operations defined for the business process – see Fig. 4.

The business rules were defined in DRL language and stored in JBoss engine. Each business rule was given a unique name:
- Rule 1 – Reduce shipping cost for gold customers.
- Rule 2 – Reduce shipping cost for platinum customers.

An example of rule definition in DRL language is shown below:

```
rule "Reduce shipping cost for gold customers"
when
  order: Order(shippingCost > 0)
  customer: Customer(category == CustomerCategory.GOLD)
then
  order.setShippingCost(order.getShippingCost() * 0.6f);
end
```

Business rules should be fired in strictly defined moments of application execution. Rule 1 and Rule 2 should be fired after execution of *calculateShippingCost* method, but only if the method was invoked inside *processOrder* flow. Both rules modify the value returned by the *calculateShippingCost* method basing on specific customer information. Following examples present how to define activation event, and a link between application business logic and business rules in proposed DSL:

```
event ShippingCostCalculation isExecutionOf method
   calculateShippingCost
   in OrderProcessingService withParams (Order)
end
link CustomizeShippingCost
  fires "*shipping cost*" after ShippingCostCalculation
  requires customer newOrder active while OrderProcessing
end
```

Business rules (Rule 1, Rule 2) are identified based on part of their names ("*shipping cost*" regular expression).

The DSL definition is automatically transformed to AspectJ code. The following code presents result of such transformation:

```
package pl.wroc.pwr.casestudy.aspects;
import pl.wroc.pwr.casestudy.domain.Customer;
...
import org.drools.StatelessSession;
...
public aspect CustomizeShippingCostAspect
  percflow(execution(
```

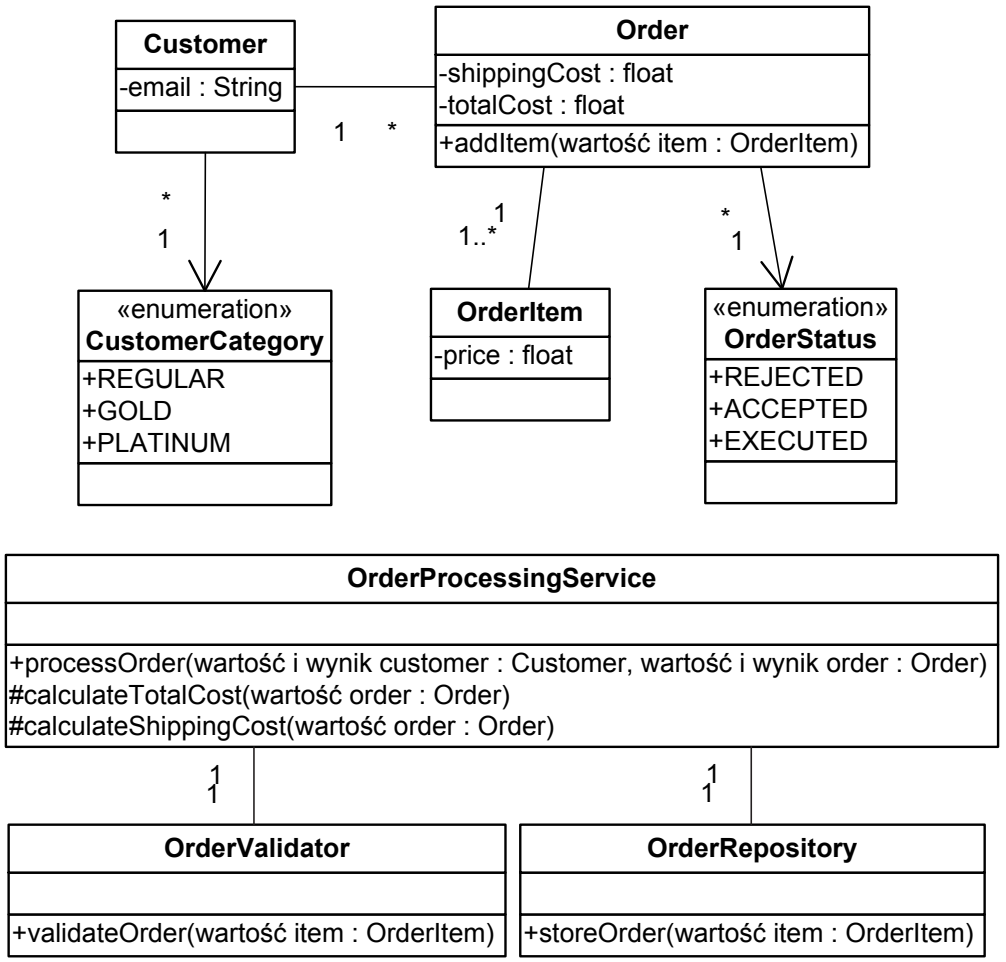


Figure 3. Data model and business logic layer for considered example

```
  void OrderProcessingService.processOrder(
                                  Customer, Order))) {
   private Customer customer;
   private Order newOrder;
   private int capturedFacts = 0;
   private static int getCapturedFacts() {
    if (CustomizeShippingCostAspect.hasAspect()) {
      return
      CustomizeShippingCostAspect.aspectOf().capturedFacts;
    } else {
      return -1;
}}
before(Customer customer, Order newOrder):
  execution(void OrderProcessingService.processOrder
  (Customer, Order)) && args (customer, newOrder)
    && if (getCapturedFacts() == 0) {
      this.customer = customer;
      this.newOrder = newOrder;
      this.capturedFacts++;
  }
pointcut shippingCostCalculationPointcut() :
  execution(
  void OrderProcessingService.calculateShippingCost(Order))
  && cflow(execution(
  void OrderProcessingService.processOrder
  (Customer, Order))) && if (getCapturedFacts() == 1);
  after() : shippingCostCalculationPointcut () {
  RuleAgent agent
          = RuleAgent.newRuleAgent("config.properties");
  RuleBase ruleBase = agent.getRuleBase();
  StatelessSession session = ruleBase.newStatelessSession();
  session.setAgendaFilter(
    new RuleNameMatchesAgendaFilter("*shipping cost*"));
  try {
    session.execute(new Object[]{customer, newOrder});
  } catch (ConsequenceException exc) {
    Throwable originalException = exc.getCause();
    if (originalException instanceof RuntimeException){
```

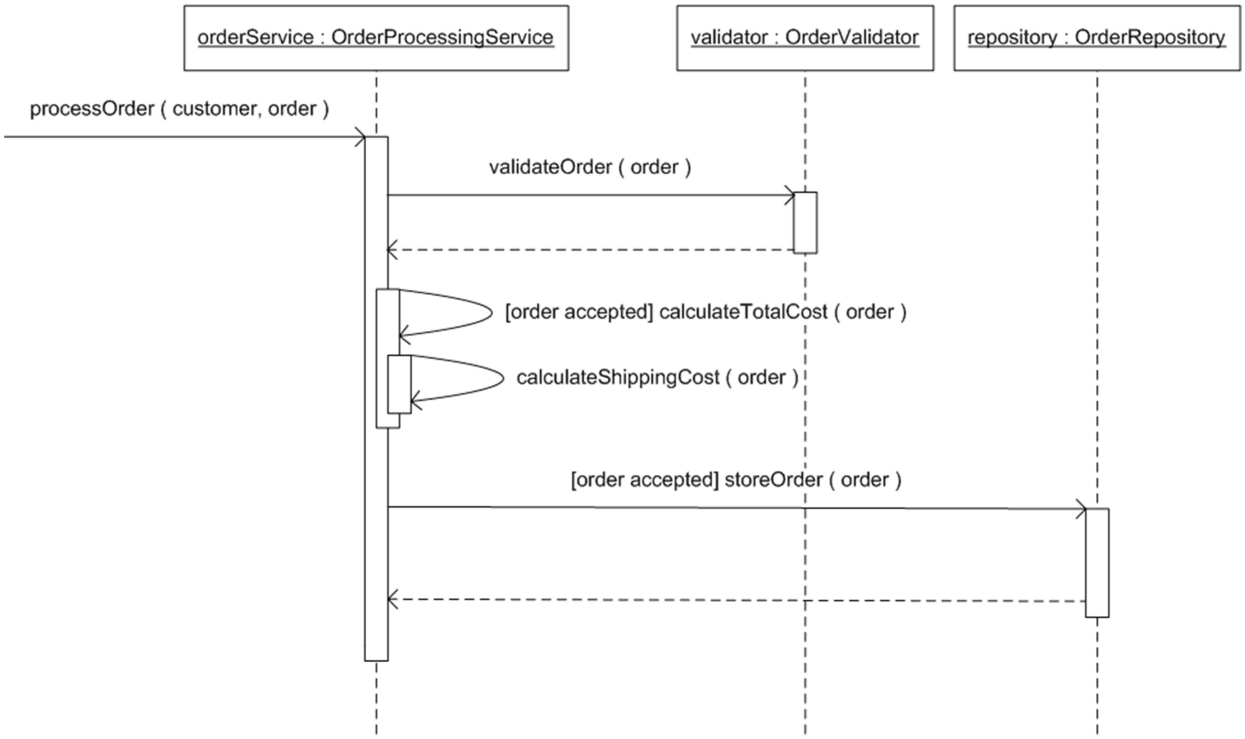


Figure 4. Order processing realization

```
      throw (RuntimeException) originalException;
   } else {
      throw exc;
}}}}
```

4.2. Example 2

The second example considers business rules for cross and circle game. For the game following rules were identified:

- Rule 1: Board size is set to 3 x 3; at the beginning all fields are free.
- Rule 2: The game is for two players: one uses circle and the other – cross symbol.
- Rule 3: Players set moves in turns.
- Rule 4: Player should place his/her symbol on a free field.
- Rule 5: Game is over if someone places 3 the same symbols in vertical, horizontal or diagonal line – this player is the winner.
- Rule 6: Game is over where there is no free place – nobody wins.

Based on business rules the class diagram presenting business terms and facts was elaborated – see Fig. 5.

In the Table 1 we consider possible ways of implementation above mentioned business rules; the first alternative is to implement business rules directly in object-oriented program; the second – to implement them in JBoss rule engine and invoke using intermediate layer. The last column in the table presents some comments, and the proposed solution.

The sequence diagram presented in Fig. 6. shows a possible implementation of message passing (main flow of events) after object-initialization before applying proposed architecture. The places when to invoke particular business rules are marked there by UML notes.

Eventually, only the business rules 4–6 were implemented within proposed architecture. Below there is an example definition of Rule 4 written a in DLR language that is accepted by JBoss engine. The rule causes an exception when the condition is evaluated to true.

```
rule "Rule4"
when
  $move: Move()
  $board: Board()
```

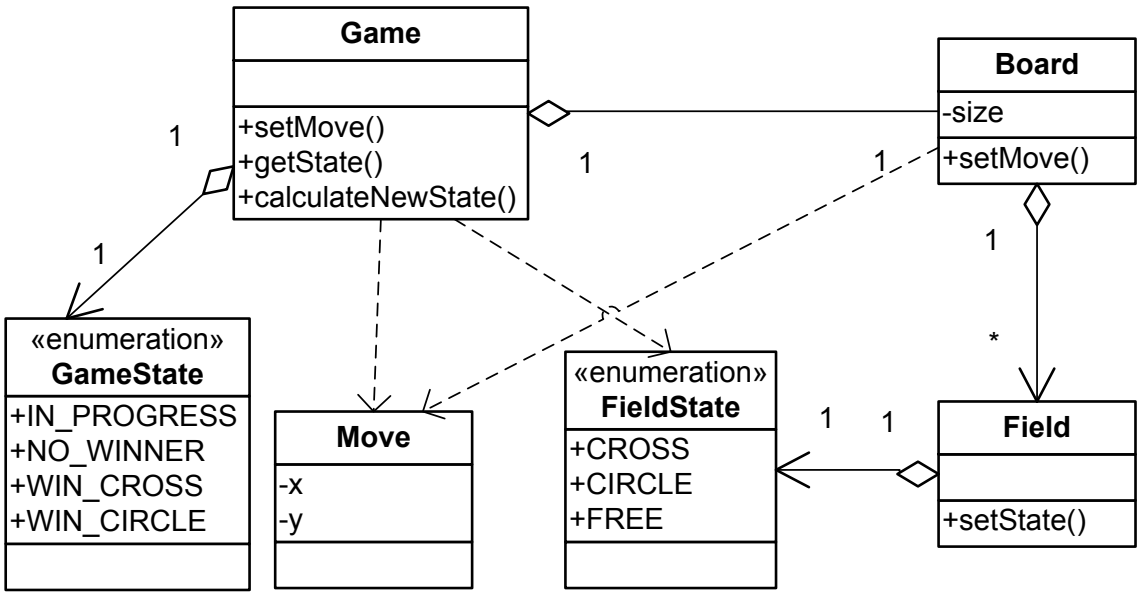


Figure 5. Terms and facts for cross and circle game

```
  eval($board.getFieldState($move) == FieldState.CROSS ||
       $board.getFieldState($move) == FieldState.CIRCLE)
then
  throw new IllegalStateException("Place must be free");
end
```

The activation event definition in DSL language as well as link definition for Rule 4 has the following form:

```
event setMoveEv isExecutionOf
  method setMove in Board asFact boardSetMoveEv
  withParams (Move asFact moveSetMoveEv FieldState
                              asFact fieldStateSetMoveEv)
end
link setMoveEv_Rule4
   fires "Rule4" before setMoveEv
   requires moveSetMoveEv boardSetMoveEv
end
```

The activation event for Rule 4 is an invocation of *setMove* method. The parameters of the method are further passed to business rule engine. Rule 4 should be evaluated before *setMove* method is called – what is expressed in link definition, and the rule itself needs two parameters to work on.

Based on DSL specification AspectJ code was generated. In the code activatation events are represented by pointcuts while links by advices:

```
public aspect setMoveEv_Rule4Aspect {
pointcut setMoveEvPointcut(Move moveSetMoveEv,
                                  Board boardSetMoveEv):
execution (void Board.setMove(Move, FieldState))
&& args (moveSetMoveEv, FieldState)
&& target (boardSetMoveEv);
before(Move moveSetMoveEv, Board boardSetMoveEv):
  setMoveEvPointcut (setMoveEv, boardSetMoveEv) {
  StatelessSession session
         = AspectHelper.ruleBase.newStatelessSession();
  session.setAgendaFilter(
             new RuleNameMatchesAgendaFilter("Rule4"));
  try {
    session.execute(new Object[]{moveSetMoveEv,
                                       boardSetMoveEv});
  } catch (ConsequenceException exc) {
    Throwable originalException = exc.getCause();
    if (originalException instanceof RuntimeException) {
      throw (RuntimeException) originalException;
   } else {
     throw exc;
}}}}
```

The rules can be checked by JUnit tests. Below unit test for Rule 4 is presented:

```
@Test(expected=IllegalStateException.class)
public void testRule4() {
  System.out.println("==Rule 4 - free place==");
  Game service = new Game();
  Move move = new Move();
  move.setX(1);
```

Table 1. Possible ways of business rules implementaiton for cross and circle game

Rule No.	Classical OO implementation	Proposed architecture	Comments
Rule 1	Appropriate constructors	Method invocation event (for constructors)	Initializing objects within JBoss is possible, however looks strange; classical OO implementation is used
Rule 2	Implemented at GUI level. Assured by terms definition	Assured by terms definition	Classical OO implementation is used
Rule 3	Implemented at GUI level	At least last move must be remebered; Method invocation event (*setMove* method)	To simplify the class structure classical OO implementation is used
Rule 4	Either *setMove* method returns *bool* value (true when place is free) or *setMethod* throws an exception when place is occupied	Method invocation event for *setMove* method; an exception is thrown when place is occupied	Proposed architecture is used
Rule 5 Rule 6	After *setMove* method the state of a game is recalculated; the interface asks a game for its new state	Method invocation event either for *setMove* method or for *calculateNewState* method; the interface asks a game for its new state	Proposed architecture is used. The new state of a game might be calculated after *setMove* method, what is not easly to observe. To increase readibility the empty *calculateNewState* method is provided for *Game* class – its invocation fires business rule 5 validation

```
  move.setY(1);
  service.setMove(move, FieldState.CROSS);
  service.setMove(move, FieldState.CIRCLE);
}
```

5. Related works

Other researchers have also noticed relationship between crosscutting nature of business rules and aspect-oriented paradigm. In [2] authors analyze if domain knowledge can be treated as one of the system's aspects that should be developed and maintained independently of other concerns. In [8] author investigates how to move implementation of business rules from core business classes to aspects. Two business rules for a system that supports transferring funds between bank accounts were defined and implemented in AspectJ language. Conducted analysis is neither thorough nor systematic but shows that aspect-oriented programming language can support implementation of business rules in object-oriented systems. In [1] an approach for expressing business rules at a high level of abstraction is presented. A high-level business rule language and high-level business rule connection language were defined. Proposed solution is similar to ours because it uses aspect-oriented paradigm to encapsulate source code that connects business rules with the core application. Each high-level rule is automatically mapped to a java class, but only inferences and computations are supported. JAsCo [6] aspect-oriented language is used for low-level implementation of connections between rules and application. For each rule connection specification an aspect bean and a connector are generated. Our approach differs from that one presented in [1] mainly because of different technologies (JBoss

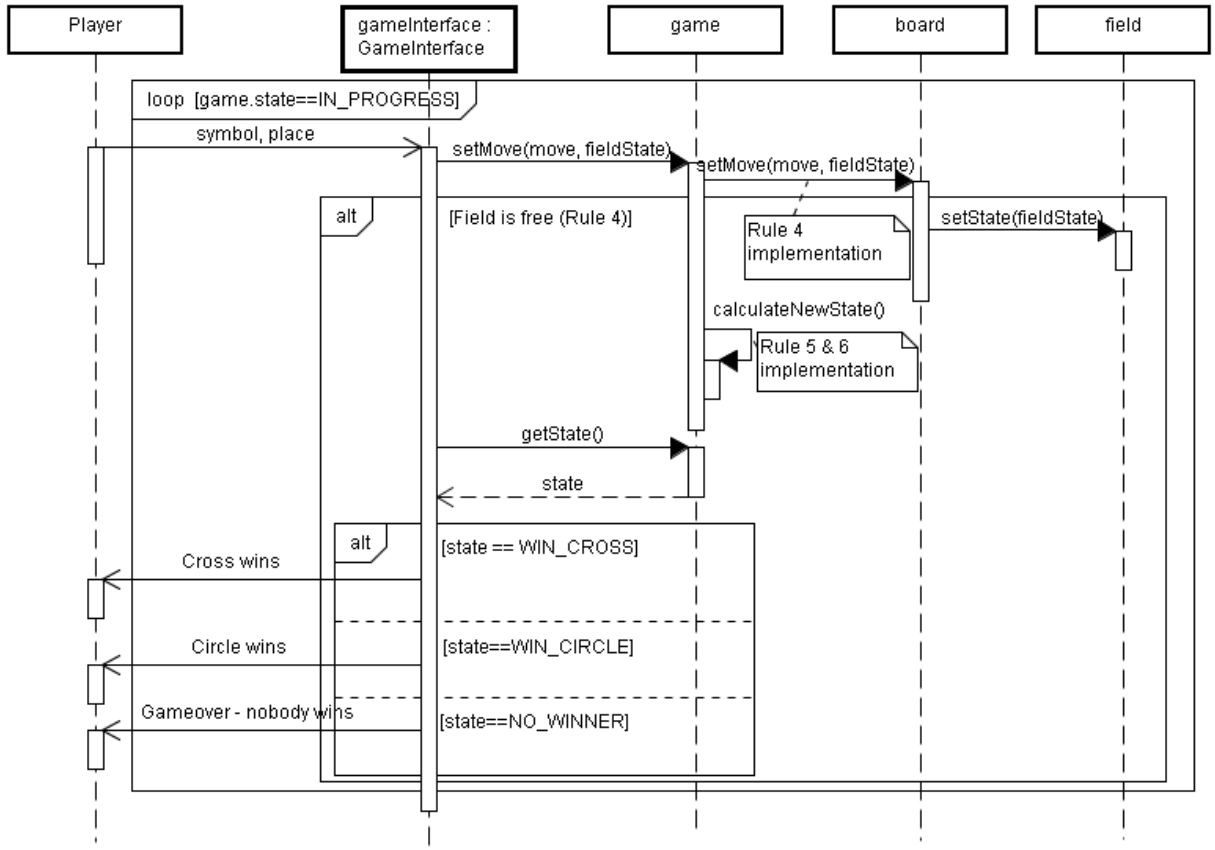


Figure 6. Sequence diagram for game scenario

instead of pure Java) and languages (AspectJ instead of JAsCo) used in a proof-of-concept implementation of the integration layer. Moreover, our DSL for integration layer is more flexible and expressive than the high-level business rule connection language proposed in [1]. It supports all kinds of business rules, allows to more precise activation event's context definition and offers better support for capturing business objects within business processes realizations.

6. Conclusions

Authors have proposed to use a specific DSL for describing dependencies between application business layer and business rules. At that moment only two types of events that result in a business rule invocation are identified (method call and attribute change). Introduction of a new event kind must be followed with extension of both, DSL syntax and DSL-to-code transformations.

Applying proposed DSL for the integration layer has many advantages. It allows to define connections between rules and business process at higher abstraction level in a declarative way. The syntax is easy and very flexible. The proof-of-concept implementation proved that the reduction above 70% in source code line numbers is possible. The solution is platform independent, so – if something changes at implementation level it will only have influence on model-to-code transformations. The transformations are complete in the sense that obtained aspect definitions need not to be changed by programmers.

The main disadvantage of DSL is that to apply it successfully you need to know the business classes, relationships among them, the semantics of their methods and the interactions among instances. Therefore, the obvious direction of further research is a formalization of business rules and business processes, that allow to abstract from their concrete implementations.

References

[1] M. A. Cibrán and M. D'Hondt. A slice of MDE with AOP: Transforming high-level business rules to aspects. In J. Smith, editor, *MoDELS*, pages 170–184, 2006.

[2] M. D'Hondt and T. D'Hondt. Is domain knowledge an aspect? In *Proceedings of the Workshop on Object-Oriented Technology*, pages 293–294, London, UK, Springer-Verlag, 1999.

[3] B. V. Halle. *Business Rules Applied – Building Better Systems Using the Business Rules Approach*. Wiley, 2002.

[4] E. F. Hill. *Jess in Action: Java Rule-Based Systems (In Action Series)*. Manning Publications, 2003.

[5] ILOG JRules. http://www.ilog.com/products/jrules/.

[6] JAsCo language documentation. http://ssel.vub.ac.be/jasco/.

[7] JBoss rules. http://www.jboss.com/products/rules.

[8] R. Laddad. *AspectJ in Action. Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[9] OpenArchitectureWare user guide. http://www.openarchitectureware.com/staticpages/index.php/documentation.

[10] R. Ross. The business rules manifesto. http://www.businessrulesgroup.org/brmanifesto.htm, 2003.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

A Case Study on Behavioural Modelling of Service-Oriented Architectures

Marek Rychlý*

*Department of Information Systems, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic

rychly@fit.vutbr.cz

Abstract

Service-oriented architecture (SOA) is an architectural style for software systems' design, which merges well-established software engineering practices. There are several approaches to describe systems and services in SOA, the services' derivation, mutual cooperation to perform specific tasks, composition, etc. In this article, we introduce a new approach to describe behaviour of services in SOA, including behaviour of underlying systems of components, which form the services' implementation. The behavioural description uses the process algebra π-calculus and it is demonstrated on a case study of a service-oriented architecture for functional testing of complex safety-critical systems.

1. Introduction

Service-oriented architecture (SOA) is a well-established architectural style for aligning business and IT architectures. It is a complex solution for analysis, design, maintaining and integration of enterprise applications that are based on services. It represents a model in which functionality is decomposed into small, distinct units, "services", which can be distributed over a network and can be combined together and reused to create business applications [10]. A system that applies SOA can be described at three levels of abstraction: as a system of business processes, services, and components.

At the first level, the system is described as a hierarchically composed business process, where each decomposable process (at each level of the composition) represents a sequence of steps in accordance with some business rules leading to a business aim.

The business processes or their parts are implemented by *services*, which are defined as autonomous platform-independent entities enabling access to their capabilities via their interfaces. *Business services* encapsulate distinct sets of business logic, *utility services* provide generic, non-application specific and reusable functionality, and *controller services* act as parent services to service composition members and ensure their assembly and coordination to the execution of the overall business task [10].

Every service can be implemented as a *component-based system* (CBS) with well-defined structure and description of its evolution for the benefit of the implementation. Then, *components* are self contained entities, parts of component-based systems accessible through well-defined *interfaces* and interconnected and communicating via *bindings* of these interfaces. *Primitive components* are realised directly, beyond the scope of architecture description (they are "black-boxes"), while *composite components* are decomposable on systems of subcomponents at the lower level of architecture description (they are "grey-boxes").

1.1. Motivation

There are several approaches to describe information systems and services in service-oriented architecture [1, 19]. Those approaches cover the whole development process from an analysis where individual services are derived from user requirements (usually represented by a system of business processes) to an implementation, which uses particular technologies implementing the services (e.g. Web Services). During this process, developers have to deal with description of a mutual cooperation of services to perform specific tasks, their composition, deployment, etc.

However, current approaches to service-oriented architecture design usually end up at the level of individual services. They do not describe underlying systems of components, which form design of individual services as component-based software systems with well-defined interfaces and behaviour.

This article introduces a development process which includes design of service-oriented architecture as well as description of underlying component-based systems. Structure of the service-oriented architecture and the component-based systems is depicted by UML-based models in a logical view, while their behaviour is formally described by means of process algebra π-calculus in a process view, with a focus on particular features such as dynamic reconfiguration and component mobility[1] in aspects of SOA. The proposed development process is illustrated on a case study of an environment for functional testing of complex safety-critical systems.

1.2. Structure of the Article

The remainder of this article is organised as follows. The case study is introduced in Section 2 and its design is described in more detail in Section 2.1 as a service-oriented architecture and in Section 2.2 as an underlying component-based system.

In Section 3, we briefly describe the π-calculus to provide formal basis, which is used later for behavioural modelling of services in the service-oriented architecture in Section 4 and for behavioural modelling of components of the component-based system in Section 5. In Section 6, the formal description of behaviour of the services and components is utilised for verification and model checking.

In Section 7, the proposed approach is discussed and briefly compared with current approaches relevant to our subject. To conclude, in Section 8, we summarise the contribution of this article and outline the future work.

2. Case Study

As a case study, we adopt specification of a SOA for functional testing of complex safety-critical systems, more specifically *a testing environment of a railway interlocking control system*, which has been described in [9]. The environment allows to distribute and run specific tests over a wide range of different testing environments, varying in their logical position in the system's architecture.

The testing environment is described as a composition of a tester and a set of external system simulators. The *external system simulators* totally or partially represent and simulate a tested environment interacting with *system under testing* (SUT), e.g. a behaviour of field objects (points, track circuits, coloured signals, etc.). The *tester* automatically executes specific tests that are coded in *test scripts* and coordinates the SUT via a *man machine interface* (MMI) and the external system simulators. The SUT is represented by the *computer based control system* (CBCS), running the *control software*, interacting with operators by means of the MMI and monitoring or controlling *external systems* of rail yards by means of sensors or actuators, which are accessible via *external systems interface*. Each *rail yard* has its own instance

[1] The *dynamic reconfiguration* represents creation, destruction and updating of components and their interconnections during the systems' run-time, while the *component mobility* allows creation of copies of components and changes of their context.

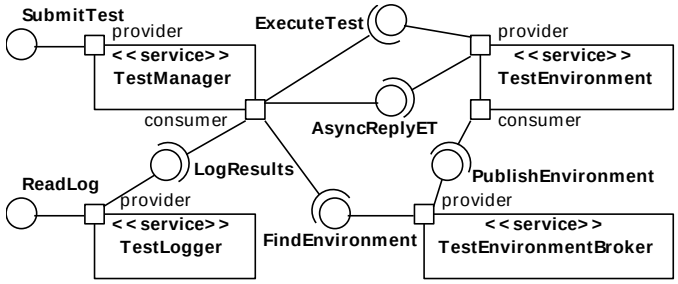


Figure 1. Services of the testing environment and their interfaces (for notation, see [19])

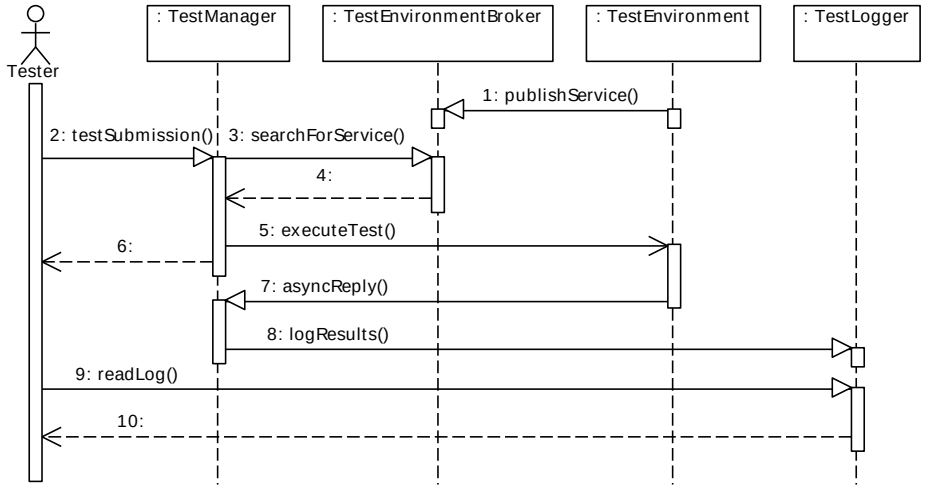


Figure 2. The choreography of services in the testing environment

of the testing environment with specific sensors and actuators where assigned tests are automatically executed. For detailed description, see [9].

To implement a system for distribution and execution of the tests over various instances of the testing environments, [9] proposes to use SOA. The system consists of a test manager, which is able to receive a test script and execute it in an instance of the testing environment. Available testing environments are registered by a broker and provided to the test manager at its request.

2.1. Service Identification

From the description of the testing environment and the system's architecture, the following tasks can be identified as invocations of services: "Submit Test", "Execute Test", "Log Results", "Read Log", "Publish Environment", and "Find Environment". The tasks can be implemented by the following business (entity) services, as it is described in Figure 1: `TestManager`, `TestEnvironment`, `TestEnvironmentBroker`, and `TestLogger`.

At first, service `TestManager` receives a test script from a tester via its interface `SubmitTest`. Then, it calls `FindEnvironment` of service `TestEnvironmentBroker` to search for a testing environment that would be suitable for the test script. The broker, which has previously accepted a registration request from a specific service `TestEnvironment` via its interface `PublishEnvironment`, provides `TestManager` with a reference to the registered service as a return value of the call of `FindEnvironment`.

After that, service `TestManager` passes the test script to the referred service `TestEnvironment` via its interface `ExecuteTest`. When the test script is finished, service `TestEnvironment` forwards its results back to service `TestManager`, which logs the results via `LogResults` of service `TestLogger`. Those results can be viewed later via `ReadLog`, which is provided by service `TestLogger` to the tester.

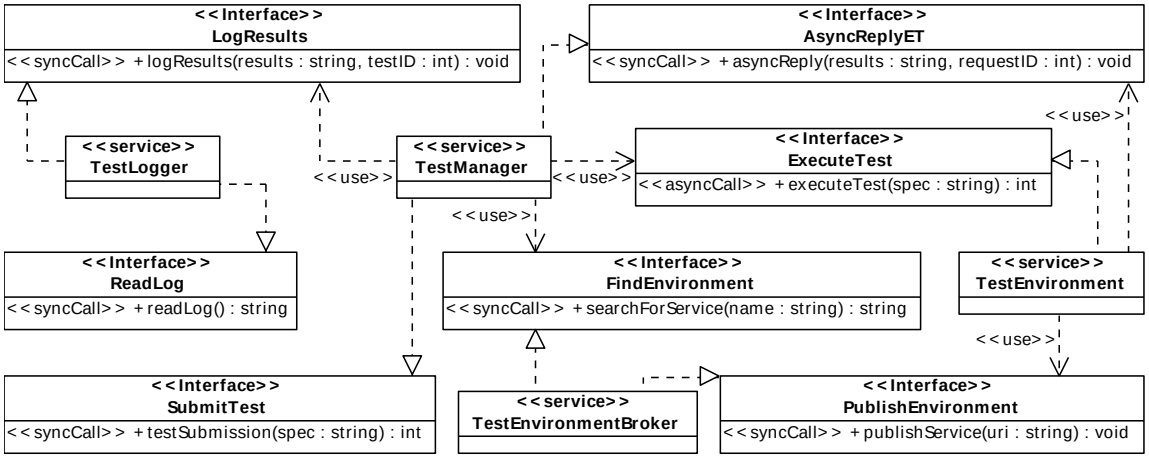


Figure 3. Services of the testing environment as UML classes

Figure 2 shows a choreography of the services as an UML sequence diagram. Detailed description of the services as classes and their interfaces with relevant stereotypes is described in the UML class diagram in Figure 3. Service `TestEnvironment` is invoked asynchronously via `ExecuteTest`, i.e. a reply corresponding to the request will be returned later via the service's interface `AsyncReplyET`.

2.2. Component-Based System

Railway interlocking control systems are safety-critical systems and can be described as component-based systems [3]. A testing environment of such systems has to interact with the systems' components, as it is described in Section 2. For that reason, a part of the testing environment, which is directly connected to a system under testing (via the external systems simulators), has character of a component neighbouring to the system and can be described as CBS.

Figure 4 describes composite component testEnvironment, which represents service `TestEnvironment` from Section 2.1. The used notation is based on our component model [17, 18] (it is not standard UML), whose detailed description is out of the scope of this article. However, in this section, we try to outline the main ideas and informally describe structure of the composite component and behaviour of its subcomponents controller, environment, test and output.

Component testEnvironment receives a test script via provided interface `executeTest`, which is internally processed by component controller. The script is represented by a fresh component, which does required testing after binding of its interfaces to component environment.

At first, component controller attaches the new component as a subcomponent test of component testEnvironment via its control interface `teAttachP`. Then, it binds interfaces `tInteract` and `tResult` of the new component to interface `eInteract` of component environment and interface `oResult` of component output, respectively. Finally, component test is activated via interface `startTestP` and executed with a new identifier via interface `executeWithID`. The identifier is also returned by component testEnvironment as a reply of the test script's submission.

Component test performs the test script by interacting with component environment via its interface `eInteract`. When the test script is finished, component test sends the test's results and its identifier to component output via its interface `oResult`. Then, component output notifies component controller via its interface `cDone` and forwards the results and the identifier out of the component testEnvironment via its external interface `asyncReplyET`.

After component controller is notified about the finished test script, it is able to receive and execute another test script, i.e. to attach a new component in the place of component test. Before that, component test with the old script is

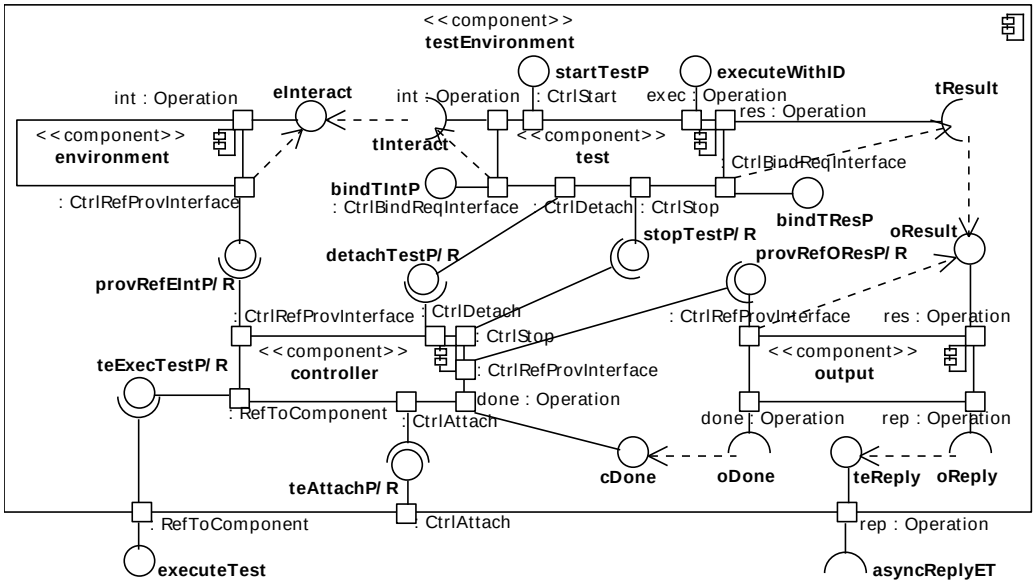


Figure 4. Composite component TestEnvironment (a specific UML-like notation)

stopped via interface `stopTestP` and detached via control interface `detachTestP`[2].

3. Formal Basis for Behavioural Modelling

To describe services in SOA and CBS in formal way, we use the *process algebra π-calculus*, known also as a *calculus of mobile processes* [16]. It allows modelling of systems with dynamic communication structures (i.e. mobile processes) by means of two concepts: processes and names. The *processes* are active communicating entities, primitive or expressed in π-calculus, while the *names* are anything else, e.g. communication links (known as "ports"), variables, constants (data), etc. Processes use names (as communication links) to interact, and they pass names (as variables, constants, and communication links) to another processes by mentioning them in the interactions. Names received by a process can be used and mentioned by it in further interactions (as communication links). For description of our approach in this article, we suppose basic knowledge of the fundamentals of the π-calculus, a theory of mobile processes, according to [20]:

- $\overline{x}\langle y\rangle.P$ is an *output prefix* that can send name y via name x (i.e. via the communication link x) and continue as process P;
- $x(z).P$ is an *input prefix* that can receive any name via name x and continue as process P with the received name substituted for every free occurrence of name z in the process;
- $P + P'$ is a *sum* of capabilities of P together with capabilities of P' processes, it proceeds as either process P or process P', i.e. when a sum exercises one of its capabilities, the others are rendered void;
- $P \mid P'$ is a *composition* of processes P and P', which can proceed independently and can interact via shared names;
- $\prod_{i=1}^{m} P_i = P_1 \mid P_2 \mid \ldots \mid P_m$ is a *multi-composition* of processes $P_1, \ldots, P_m$, for $m \geq 3$, which can proceed independently interacting via shared names;
- $(z)P$ is a *restriction* of the scope[3] of name z in process P;
- $(\tilde{x})P = (x_1, x_2, \ldots, x_n)P = (x_1)(x_2)\ldots(x_n)P$ is a *multi-restriction* of the scope of names $x_1, \ldots, x_n$ to process P, for $n \geq 2$,

[2] In the diagram in Figure 4, only these two interfaces of test are connected with controller, because the rest of the test's interfaces are used only during its nesting and their connections do not exist outside of controller component.

[3] The scope of a restriction may change as a result of interaction between processes.

– $!P$ is a *replication* that means an infinite composition of processes P or, equivalently, a process satisfying the equation $!P = P \mid !P$.

The π-calculus processes can be parametrised. A parametrised process, referred as an *abstraction*, is an expression of form $(x).P$.

When abstraction $(x).P$ is applied to argument y it yields process $P\{y/x\}$, i.e. process P with y substituted for every free occurrence of x. Application is a destructor of the abstraction. We can define two types of application: pseudo-application and constant application.

Pseudo-application $F\langle y\rangle$ of abstraction $F \stackrel{def}{=} (x).P$ is an abbreviation of substitution $P\{y/x\}$. On the contrary, the *constant application* is a real syntactic construct, which allows to reduce a form of process $K\lfloor y\rfloor$, sometimes referred as an *instance* of process constant K, according to a *recursive definition* of process constant $K \triangleq (x).P$. The result of the reduction yields process $P\{y/x\}$.

4. Behavioural Modelling of Services

In this section, we describe behaviour of the services in the testing environment. Behaviour of services `TestManager`, `TestEnvironmentBroker`, `TestEnvironment`, and `TestLogger` can be described by means of π-calculus process abstractions TM, TEB, TE, and TL, respectively. These process abstractions use names st, pe, fe, et, ar, lr, and rl as representations of the services' interfaces `SubmitTest`, `PublishEnvironment`, `FindEnvironment`, `ExecuteTest`, `AsyncReplyET`, `LogResults`, and `ReadLog`, respectively.

According to the description of `TestEnvironment` in Section 2.1, process abstraction TM describing behaviour of service `TestManager` is defined as follows:

$$TM \stackrel{def}{=} (st, fe, lr).(s)(TM_{st}\lfloor st, fe, s\rfloor \mid TM_{ar}\lfloor lr, s\rfloor)$$

$$TM_{st} \triangleq (st, fe, s).st(test, ret).(r, r')\ (\overline{fe}\langle r\rangle.r(et', ar').\overline{et'}\langle test, r'\rangle.(r'(id).\overline{ret}\langle id\rangle \mid \overline{s}\langle ar'\rangle \mid TM_{st}\lfloor st, fe, s\rfloor))$$

$$TM_{ar} \triangleq (lr, s).s(ar')ar'(res, id).\overline{lr}\langle res, id\rangle \mid TM_{ar}\lfloor lr, s\rfloor$$

where st, fe, and lr are names representing the service's interfaces and subsequently processed by constant applications of TM_{st} and TM_{ar}.

Constant application $TM_{st}\lfloor st, fe, s\rfloor$ receives a pair of names $(test, ret)$ from a client via name st. In the pair, name $test$ represents a submitted test script and name ret will be used later to send a return value to the client. Then, a request for a testing environment is sent via name fe and the environment as a reply is received via name r. Name et', which represents an interface `ExecuteTest` of the environment, is used to send $test$. Name id, which is received as a return value, is forwarded to the client, while name ar' is sent via shared name s into process constant TM_{ar}. Constant application $TM_{ar}\lfloor lr, s\rfloor$ receives name ar' via shared name s. After the test script is finished, name ar' is used to receive the test's result res and its id. These names, as a pair (res, id), are immediately sent via name lr.

Process abstraction TEB, which describes behaviour of service `TestEnvironmentBroker`, is defined as follows:

$$TEB \stackrel{def}{=} (pe, fe).(q)(TEB_{pub}\lfloor q, pe\rfloor \mid TEB_{find}\lfloor q, fe, pe\rfloor)$$

$$TEB_{pub} \triangleq (t, pe).pe(i, d).(t')(\overline{t}\langle t', i, d\rangle \mid TEB_{pub}\lfloor t', pe\rfloor)$$

$$TEB_{find} \triangleq (h, fe, pe).h(h', i, d).(TEB_{find}\lfloor h', fe, pe\rfloor \mid (\overline{fe}\langle i\rangle.\overline{pe}\langle i, d\rangle + d))$$

where pe and fe are names representing the service's interfaces `PublishEnvironment` and `FindEnvironment`, respectively, and subsequently processed by the constant applications of TEB_{pub} and TEB_{find}. By the composition of their constant applications with shared name q, process abstraction TEB implements basic operations on a simple queue (i.e. a First-In-First-Out (FIFO) data structure).

The application of process constant TEB_{pub} receives a pair of names (i, d) via name pe and creates a new name t'. Then, it proceeds as a composition of a constant application of $TEB_{pub}\lfloor t', pe\rfloor$, which handles future requests, and process $\overline{t}\langle t', i, d\rangle$, which enqueues the received pair (i, d) by sending them via name t, which is *the current tail of the queue*, together with name t', a new tail of the queue used in the future requests.

The application of process constant TEB_{find} dequeues a front item of the queue as a triple of names (h', i, d) via name h, which is *the current head of the queue*. Then, it proceeds as a composition of a constant application of $TEB_{find}\lfloor h', fe, pe\rfloor$, which handles future requests, and a sum of capabilities of process $\overline{fe}\langle i\rangle.\overline{pe}\langle i, d\rangle$, which provides name i as an interface for potential service requesters and enqueues it back to the queue via name pe, and process d, which, after receiving a name via name d, allows to remove the interface and does not provide it to potential service requesters any more.

Behaviour of service `TestEnvironment` is described as process abstraction TE and defined as follows:

$$TE \stackrel{def}{=} (et, ar, pe).TE_{init}\langle et, ar, pe\rangle.TE_{impl}\langle et, ar\rangle$$

$$TE_{init} \stackrel{def}{=} (et, ar, pe).\overline{pe}\langle et, ar\rangle$$

$$TE_{impl} \stackrel{def}{=} (et, ar).(s_0, s_1, ar^s, et^g)$$
$$(\overline{ar^s}\langle ar\rangle \mid (d, t)(\overline{et^g}\langle t\rangle.t(p).Wire\lfloor et, p, d\rfloor) \mid TE_{comp}\langle s_0, s_1, et^g, ar^s\rangle)$$

where et, ar, and pe are names representing the service's interfaces `ExecuteTest`, `AsyncReplyET`, and `PublishEnvironment`, respectively. Initialisation of the service is described as process abstraction TE_{init}, which sends the service's interfaces represented by names et and ar via name pe (i.e. publishes the corresponding interfaces via interface `PublishEnvironment`). After the initialisation, names et and ar are processed by pseudo-application $TE_{impl}\langle et, ar\rangle$, which describes behaviour of a component-based system implementing the service (service `TestEnvironment` is implemented as the component-based system, see Section 2.2). Process abstraction TE_{comp} will be described later, in Section 5.

Finally, process abstraction TL, which describes behaviour of service `TestLogger`, is defined as follows:

$$TL \stackrel{def}{=} (lr, rl).(s)(TL_{lr}\lfloor lr, s\rfloor \mid TL_{rl}\lfloor rl, s\rfloor)$$

$$TL_{lr} \triangleq (lr, t).lr(res, id).(t')(\overline{t}\langle t', res, id\rangle \mid TL_{lr}\lfloor lr, t'\rfloor)$$

$$TL_{rl} \triangleq (rl, h).h(h', res, id).rl(ret).\overline{ret}\langle res, id\rangle.TL_{rl}\lfloor rl, h'\rfloor$$

where lr and rl are names representing the service's interfaces `LogResults` and `ReadLog`, respectively, and subsequently processed by the applications of process constants TL_{lr} and TL_{rl}. The process abstraction TL uses an internal queue to store log results. The queue is accessed in process constants TL_{lr} and TL_{rl} via name h for a head of the queue and name t for a tail of the queue, respectively. At the beginning, h and t are identical to name s in process abstraction TL.

Constant application $TL_{lr}\lfloor lr, t\rfloor$ receives a pair of names (res, id) via name lr, which will be added into the internal queue. It creates name t' (as a new tail of the queue) and sends via t' the pair of names (res, id) and name t (an original tail of the queue). Concurrently, the process proceeds as the application of process constant TL_{lr} with name t' (the new tail of the queue).

Constant application $TL_{rl}\lfloor rl, h\rfloor$ receives a first queued item via name h (from a head of the queue). This item contains a pair of names (res, id) and name h' (a new head of the queue). After the pair of names (res, id) is requested via name rl, it is sent via name ret as a reply and the process proceeds as the application of process constant TL_{rl} with name h' (the new head of the queue).

Behaviour of the whole system of the interconnected services can be described as process abstraction $System$, which provides names st and rl representing interfaces `SubmitTest` and `ReadLog`, respectively, and which is defined as follows:

$$\begin{aligned} System \stackrel{def}{=}\ & (st, rl).(et, ar, lr, pe, fe) \\ & (TM\langle st, fe, lr\rangle \mid TE\langle et, ar, pe\rangle \mid TL\langle lr, rl\rangle \mid TEB\langle pe, fe\rangle) \end{aligned}$$

5. Behavioural Modelling of the Component-Based System

All processes, which represent behavioural descriptions of individual services, have been described completely, except for process abstraction TE of service `TestEnvironment` implemented as a component-based system with behaviour described by pseudo-application $TE_{comp}\langle s_0, s_1, ar^s, et^g\rangle$. In this section, we describe behaviour of primitive components controller, environment, test, and output, as process abstractions Ctr, Env, $Test$, and Out, respectively, and their parent composite component testEnvironment, as process abstraction TE_{comp}.

5.1. Core Behaviour of Primitive Components

Core behaviour of primitive components output and controller can be defined as process abstractions Out_{core} and Ctr_{core}, respectively, as follows:

$$Out_{core} \stackrel{def}{=} (p_{oResult}, r_{oDone}, r_{oReply}).Out'_{core}\lfloor p_{oResult}, r_{oDone}, r_{oReply}\rfloor$$

$$\begin{aligned} Out'_{core} \stackrel{\Delta}{=}\ & (p_{oResult}, r_{oDone}, r_{oReply}).p_{oResult}(res, id).\overline{r_{oDone}}\langle id\rangle. \\ & (\overline{r_{oReply}}\langle res, id\rangle \mid Out'_{core}\lfloor p_{oResult}, r_{oDone}, r_{oReply}\rfloor) \end{aligned}$$

$$\begin{aligned} Ctr_{core} \stackrel{def}{=}\ & (p_{cDone}, p_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, \\ & r_{provRefORes}).Ctr'_{core}\lfloor p_{cDone}, p_{teExecTest}, \\ & r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}\rfloor \end{aligned}$$

$$\begin{aligned} Ctr'_{core} \stackrel{\Delta}{=}\ & (p_{cDone}, p_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, \\ & r_{provRefORes}).p_{teExecTest}(ts, ret).ts(r'_{stopTest}, r'_{startTest}, r', p'). \\ & \overline{r_{stopTest}}.\overline{r_{detachTest}}.\overline{r_{teAttach}}\langle r'_{stopTest}, r'_{startTest}, r_{detachTest}\rangle. \\ & r'(p'_{bindTInt}, p'_{bindTRes}).p'(p'_{provRefExecuteWithID}).(ret')(\\ & \overline{r_{provRefEInt}}\langle ret'\rangle.ret'(eInteract).\overline{p'_{bindTInt}}\langle eInteract\rangle. \end{aligned}$$

$$\overline{r_{provRefORes}}\langle ret'\rangle.ret'(oResult).\overline{p'_{bindTRes}}\langle oResult\rangle.$$

$$\overline{p'_{provRefExecuteWithID}}\langle ret'\rangle.ret'(p'_{executeWithID}).\overline{r'_{startTest}}.$$

$$((id)\overline{ret}\langle id\rangle.\overline{p'_{executeWithID}}\langle id\rangle.\overline{id} \mid p_{cDone}(id').id'.$$
$$Ctr'_{core}\lfloor p_{cDone}, p_{teExecTest}, r_{teAttach}, r_{detachTest},$$
$$r'_{stopTest}, r_{provRefEInt}, r_{provRefORes}\rfloor))$$

where the components' provided or required interfaces are represented by names $p_{...}$ or $r_{...}$, respectively, without the last character ($\ldots P/R$, see Figure 4).

Process abstraction Out_{core} is defined as the constant application of Out'_{core}. It receives a pair of names (res, id) via name $p_{oResult}$ representing interface `oResultP`. Then, id is sent via name r_{oDone} (interface `oDoneR`) and (res, id) is forwarded via name r_{oReply} (interface `oReplyR`) out of the composite component.

Process constant Ctr'_{core}, which is applied by process abstraction Ctr_{core}, receives a pair of names (ts, ret) via name $p_{teExecTest}$. Moreover, via name ts, the constant receives also names $r'_{stopTest}$, $r'_{startTest}$, c, and indirectly also names $p'_{bindTInt}$, $p'_{bindTRes}$, and $p'_{provRefExecuteWithID}$, which represent interfaces of a new component compatible with component `test` and implementing a test script. Name ret will be used later to send an identifier of the test's results as a return value. Then, a process of an old component `test` is deactivated and detached by means of names $r_{stopTest}$ and $r_{detachTest}$. A process, which describes behaviour of the new component (i.e. the actual test script), is attached via name $r_{teAttach}$ as a subcomponent, bound via names $p'_{bindTInt}$ and $p'_{bindTRes}$, activated via name $r'_{startTest}$, and finally, it is executed via name $p'_{executeWithID}$ with a new name id (the identifier). Processing of Ctr'_{core} continues after the identical id is received via name p_{cDone}, i.e. the test script is finished and its results forwarded outside.

Core behaviour of components `environment` and `test` depends on a specific implementation of the testing environment and on a specific test script. However, for demonstrating purposes, we define process abstractions Env_{core} and $Test_{core}$:

$$Env_{core} \stackrel{def}{=} (p_{eInteract}).Env'_{core}\lfloor p_{eInteract}\rfloor$$

$$Env'_{core} \stackrel{\Delta}{=} (p_{eInteract}).p_{eInteract}(ret).((val)\overline{ret}\langle val\rangle \mid Env'_{core}\lfloor p_{eInteract}\rfloor)$$

$$Test_{core} \stackrel{def}{=} (p_{executeWithID}, r_{tInteract}, r_{tResult}).p_{executeWithID}(id).$$
$$(ret)(\overline{r_{tInteract}}\langle ret\rangle.ret(val).\overline{r_{tResult}}\langle val, id\rangle)$$

Process constant Env'_{core} receives a request from a test script via name $p_{eInteract}$ and returns a new name val as a reply. Process abstraction $Test_{core}$ receives identifier id via name $p_{executeWithID}$, sends a request to a process representing behaviour of a test environment via name $r_{tInteract}$, receives a reply and forwards it as the test's results together with id via name $r_{tResult}$.

5.2. Behaviour of a Composite Component

To assemble (sub)components into a composite component, we need to implement control actions. Components, primitive or composite, provide control interfaces for referencing their provided functional interfaces, binding their required functional interfaces (to the referred provided interfaces), and controlling their life-cycle (to start and stop the components). Moreover, each composite component provides its subcomponents with (internal) control interfaces for attaching and detaching other subcomponents, exporting their functional interfaces as the composite component's (external)

functional interfaces, and importing the composite component's (external) functional interfaces to its subcomponents.

Behaviour associated with those control actions can be described in π-calculus. At first, let us define an auxiliary constant application $Wire\lfloor x, y, d\rfloor$, which can receive a message via name x (an input) and send it via name y (an output) repeatedly till it receives a message via name d (i.e. disable processing). Then, let us assume that $Ctrl_{Ifs}\langle r_1, \ldots, r_n, p_1^s, \ldots, p_n^s, p_1, \ldots, p_m, p_1^g, \ldots, p_m^g\rangle$ represents behaviour, which is associated with binding of interfaces represented by names $r_1, \ldots, r_n$ via control interfaces represented by names $p_1^s, \ldots, p_n^s$ and referencing of interfaces represented by $p_1, \ldots, p_m$ via control interfaces represented by $p_1^g, \ldots, p_m^g$.

$$
\begin{aligned}
Wire &\triangleq (x, y, d).(x(m).\overline{y}\langle m\rangle.Wire\lfloor x, y, d\rfloor \ + \ d)\\
SetIf &\triangleq (r, s, d).s(p).(\overline{d}.Wire\lfloor r, p, d\rfloor \mid SetIf\lfloor r, s, d\rfloor)\\
GetIf &\stackrel{def}{=} (p, g).g(r).\overline{r}\langle p\rangle\\
Plug &\stackrel{def}{=} (d).d\\
Ctrl_{Ifs} &\stackrel{def}{=} (r_1, \ldots, r_n, p_1^s, \ldots, p_n^s, p_1, \ldots, p_m, p_1^g, \ldots, p_m^g).\\
&\qquad (\prod_{i=1}^{n}(r_i^d)(Plug\langle r_i^d\rangle \mid SetIf\lfloor r_i, p_i^s, r_i^d\rfloor) \mid \prod_{j=1}^{m}!GetIf\langle p_j, p_j^g\rangle)
\end{aligned}
$$

Moreover, let us assume that $Ctrl_{EI}\langle r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n\rangle$ represents behaviour of interconnections between external required and provided interfaces represented by names $r_1, \ldots, r_n$ and $p_1, \ldots, p_m$ and internal provided and required interfaces represented by names $p'_1, \ldots, p'_n$ and $r'_1, \ldots, r'_m$, respectively.

$$
\begin{aligned}
Ctrl_{EI} &\stackrel{def}{=} (r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n).\\
&\qquad \prod_{i=1}^{n}(d)Wire\lfloor r_i, p'_i, d\rfloor \mid \prod_{j=1}^{m}(d)Wire\lfloor r'_j, p_j, d\rfloor
\end{aligned}
$$

Finally, let us assume that $Ctrl_{SS}\langle s_0, s_1, a\rangle$ represents behaviour, which is associated with a component's life-cycle (s_0 for stopping and s_1 for starting the component) and attaching new subcomponents (via a).

$$
\begin{aligned}
Dist &\triangleq (p, m, r).(\overline{p}\langle m\rangle.Dist\lfloor p, m, r\rfloor \ + \ \overline{r})\\
Life &\triangleq (s_x, s_y, p_x, p_y).s_x(m).(r)(Dist\lfloor p_x, m, r\rfloor \mid r.Life\lfloor s_y, s_x, p_y, p_x\rfloor)\\
Attach &\stackrel{def}{=} (a, p_0, p_1).a(c_0, c_1, c_d)(d)\\
&\qquad (c_d(m).\overline{d}\langle m\rangle.\overline{d}\langle m\rangle \mid Wire\lfloor p_0, c_0, d\rfloor \mid Wire\lfloor p_1, c_1, d\rfloor)\\
Ctrl_{SS} &\stackrel{def}{=} (s_0, s_1, a).(p_0, p_1)(Life\lfloor s_1, s_0, p_1, p_0\rfloor \mid !Attach\langle a, p_0, p_1\rangle)
\end{aligned}
$$

With the above mentioned process abstractions and constants, behaviour of components output, environment, and test including their control parts can be defined as process abstractions Out, Env,

and $Test$, respectively:

$$\begin{aligned} Out \stackrel{def}{=}\ & (s_0, s_1, p^g_{oResult}, p^s_{oDone}, p^s_{oReply}).(p_{oResult}, r_{oDone}, r_{oReply}) \\ & (Ctrl_{Ifs}\langle p_{oResult}, p^g_{oResult}\rangle \mid Ctrl_{Ifs}\langle r_{oDone}, p^s_{oDone}\rangle \\ & \mid Ctrl_{Ifs}\langle r_{oReply}, p^s_{oReply}\rangle \mid Out_{core}\langle p_{oResult}, r_{oDone}, r_{oReply}\rangle) \end{aligned}$$

$$\begin{aligned} Env \stackrel{def}{=}\ & (s_0, s_1, p^g_{eInteract}).(p_{eInteract}) \\ & (Ctrl_{Ifs}\langle p_{eInteract}, p^g_{eInteract}\rangle \mid Env_{core}\langle p_{eInteract}\rangle) \end{aligned}$$

$$\begin{aligned} Test \stackrel{def}{=}\ & (s_0, s_1, p^g_{executeWithID}, p^s_{tInteract}, p^s_{tResult}). \\ & (p_{executeWithID}, r_{tInteract}, r_{tResult})(Ctrl_{Ifs}\langle r_{tInteract}, p^s_{tInteract}\rangle \\ & \mid Ctrl_{Ifs}\langle p_{executeWithID}, p^g_{executeWithID}\rangle \mid Ctrl_{Ifs}\langle r_{tResult}, p^s_{tResult}\rangle \\ & \mid Test_{core}\langle p_{executeWithID}, r_{tInteract}, r_{tResult}\rangle) \end{aligned}$$

Behaviour of component controller is defined as process abstraction Ctr with free names $r_{teAttach}$, $r_{detachTest}$, $r_{stopTest}$, $r_{provRefEInt}$ and $r_{provRefORes}$ representing required control interfaces of other components:

$$\begin{aligned} Ctr \stackrel{def}{=}\ & (s_0, s_1, p^g_{cDone}, p^g_{teExecTest}, \\ & \quad r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}). \\ & (p_{cDone}, p_{teExecTest})(Ctrl_{Ifs}\langle p_{cDone}, p^g_{cDone}\rangle \\ & \mid Ctrl_{Ifs}\langle p_{teExecTest}, p^g_{teExecTest}\rangle \mid Ctr_{core}\langle p_{cDone}, p_{teExecTest}, \\ & \quad r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}\rangle) \end{aligned}$$

Behaviour of composite component testEnvironemt, i.e. the implementation of the core of service `TestEnvironment`, is described as process abstraction TE_{comp}:

$$\begin{aligned} TE_{comp} \stackrel{def}{=}\ & (s_0, s_1, p^g_{executeTest}, p^s_{asyncRepltET}).(p_{executeTest}, r_{teExecTest}, \\ & p^s_{teExecTest}, r_{asyncRepltET}, p_{teReply}, p^g_{teReply}, p_{teAttach}) \\ & (Ctrl_{Ifs}\langle p_{executeTest}, p^g_{executeTest}\rangle \mid Ctrl_{Ifs}\langle r_{teExecTest}, p^s_{teExecTest}\rangle \\ & \mid Ctrl_{Ifs}\langle r_{asyncRepltET}, p^s_{asyncRepltET}\rangle \mid Ctrl_{Ifs}\langle p_{teReply}, p^g_{teReply}\rangle \\ & \mid Ctrl_{EI}\langle p_{executeTest}, r_{teExecTest}\rangle \mid Ctrl_{EI}\langle p_{teReply}, r_{asyncRepltET}\rangle \\ & \mid Ctrl_{SS}\langle s_0, s_1, p_{teAttach}\rangle \mid TE'_{comp}\langle p_{teAttach}, p^s_{teExecTest}, p^g_{teReply}\rangle) \end{aligned}$$

$$\begin{aligned} TE'_{comp} \stackrel{def}{=}\ & (p_{teAttach}, p^s_{teExecTest}, p^g_{teReply}).(s^{ctr}_0, s^{ctr}_1, s^{out}_0, s^{out}_1, s^{env}_0, s^{env}_1, \\ & \quad p^g_{cDone}, p^g_{eInteract}, p^g_{oResult}, p^g_{teExecTest}, p^s_{oDone}, p^s_{oReply}, \\ & \quad r_{detachTest}, r_{provRefEInt}, r_{provRefORes}, r_{stopTest}, r_{teAttach}) \\ & (Ctr\langle s^{ctr}_0, s^{ctr}_1, p^g_{cDone}, p^g_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest}, \\ & \quad r_{provRefEInt}, r_{provRefORes}\rangle \mid Env\langle s^{env}_0, s^{env}_1, p^g_{eInteract}\rangle \\ & \mid Out\langle s^{out}_0, s^{out}_1, p^g_{oResult}, p^s_{oDone}, p^s_{oReply}\rangle \mid (d)\overline{p_{teAttach}}\langle s^{ctr}_0, s^{ctr}_1, d\rangle \\ & \mid (d)\overline{p_{teAttach}}\langle s^{out}_0, s^{out}_1, d\rangle \mid (d)\overline{p_{teAttach}}\langle s^{env}_0, s^{env}_1, d\rangle \\ & \mid Test_{plug}\langle r_{detachTest}, r_{stopTest}\rangle \mid (d)Wire\lfloor r_{provRefEInt}, p^g_{eInteract}, d\rfloor \end{aligned}$$

$$\mid (d)Wire\lfloor r_{provRefORes}, p^g_{oResult}, d\rfloor \mid (d)Wire\lfloor r_{teAttach}, p_{teAttach}, d\rfloor$$

$$\mid (ret)(\overline{p^g_{teExecTest}}\langle ret\rangle.ret(p_{teExecTest}).\overline{p^s_{teExecTest}}\langle p_{teExecTest}\rangle)$$

$$\mid (ret)(\overline{p^g_{teReply}}\langle ret\rangle.ret(p_{teReply}).\overline{p^s_{oReply}}\langle p_{teReply}\rangle)$$

$$\mid (ret)(\overline{p^g_{cDone}}\langle ret\rangle.ret(p_{cDone}).\overline{p^s_{oDone}}\langle p_{cDone}\rangle)\ )$$

$$Test_{plug} \stackrel{def}{=} (r_{detachTest}, r_{stopTest}).(r_{detachTest} \mid r_{stopTest})$$

Process abstraction TE'_{comp}, which is applied in process abstraction TE_{comp}, creates concurrent processes given by pseudo-applications of Ctr, Out, and Env and sends their names $s_{\vec{0}}$ and $s_{\vec{1}}$ via name $p_{teAttach}$, i.e. attaches components controller, output, and environment, respectively, as subcomponents of component testEnvironment. It also interconnects names representing required and provided control interfaces of the components by means of three constant applications of $Wire$. Concurrently with the previous step, TE'_{comp} applies process abstraction $Test_{plug}$ and binds name $p_{teExecTest}$ of the pseudo-application of Ctr to name $r_{teExecTest}$ of the pseudo-application of TE_{comp}, name p_{cDone} of Ctr to name r_{cDone} of Out, and name $p_{teReply}$ of TE_{comp} to name $r_{teReply}$ of Out. The pseudo-application of process abstraction $Test_{plug}$ handles requests initiated by the pseudo-application of Ctr and received by names $r_{stopTest}$ and $r_{detachTest}$ to stop and to detach a process representing behaviour of a previous but non-existent component with a test script (e.g. a non-existent predecessor of component test).

6. System Properties and Their Verification

Formally described behaviour of services and components allow us to make simulations of the behaviour, to detect deadlocks, and to check strong and weak open bisimulation equivalences between behaviours of different services and components. This can be useful, especially to check the *test scripts*, which are processed by the *tester*, and to control the tester's behaviour and communication with other parts of the environment and with SUT (see Section 2). The wrong behaviour or the erroneous communication can cause the tests to fail and, moreover, may block future requests to the testing environment.

The behaviour formally described in the previous sections can be used for verification and model checking by means of external verification tools such as *The Mobility Workbench* (MWB, [21]) and *Another/Advanced Bisimulation Checker* (ABC, [4]). The utilisation is demonstrated by examples of interactive simulation in Section 6.1, finding deadlocks in Section 6.2, bisimulation checking in Section 6.3, and model checking in Section 6.4.

6.1. Simulation

To simulate behaviour of the system from the case study, which has been described by means of process abstraction $System$ from Section 4, we need to submit a sample test to the system, wait for its processing and finally, receive its results. Therefore, agent `Tester` is defined as follows:

```
agent Tester = (^s0,s1,pgexecuteWithID,pstInteract,
                                          pstResult,rl,st) (
  Test(s0,s1,pgexecuteWithID,pstInteract,pstResult)
  | System(st,rl) | (^ts,ret,r,p) 'st<ts,ret>
     . 'ts<s0,s1,r,p> . 'r<pstInteract,pstResult>
     . 'p<pgexecuteWithID> . ret(id1)
     . (^r2) 'rl<r2> . r2(res,id2) . 0 )
```

Agent `Tester` is a composition of the applications of agents `Test` and `System`, and an auxiliary π-calculus process (after the last composition operator). Agents `Test` and `System` represent process abstraction $System$ from Section 4

and process abstraction $Test$ from Section 5.2, respectively, with their notations adapted to MWB and ABC.

The auxiliary process submits all names of the application of agent `Test` (i.e. names `s0`, `s1`, `pgexecuteWithID`, `pstInteract` and `pstResult`) indirectly via name `st` to the application of agent `System` and receives name `id1` as a reply via name `ret`. Then, it waits for results of a test performed by the application of agent `Test`, which can be received via name `rl` of the application of agent `System`.

Behaviour of agent `Tester` can be interactively simulated in MWB by means of command "`step Tester`". However, the simulation is not transparent but demanding because of large amount of possible internal (silent) actions.

6.2. Deadlocks

A *deadlock* occurs in a π-calculus process iff the process can not perform any reduction step, i.e. the process is not responding to any action on its free names (see Section 3).

To permit concurrent processing of multiple requests, process abstractions and constants TM_{st}, TM_{ar}, TEB_{pub}, TEB_{find}, TL_{lr} TL_{rl}, Out'_{core}, and Env'_{core}, from Sections 4 and 5 use unguarded or weakly guarded recursions (i.e. guarded by unobservable prefix τ). These processes, as separate units, do not come to deadlocks, because each of them can always perform at least one reduction step[4].

Agents representing the processes from the case study have been checked for deadlocks, by means of command "`deadlocks`" in MWB. In some cases, the deadlock-checking can not be finished due to the unguarded or weakly guarded recursions (only guarded recursions are handled correctly). However, the *deadlocks have been found* in agents `TestCore`, `TestPlug`, `Wire`, `Dist`, `TE2comp`, and `TEimpl`.

Agents `TestCore`, `TestPlug`, `Wire`, and `Dist` have deadlocks in process 0, which is reachable by 1, 4, 2, and 1 commitments, respectively. These deadlocks are desired, since the agents represent process abstractions $Test_{core}$ (see Section 5.1) and process abstractions and constants $test_{plug}$, $Wire$ and $Dist$ (see Section 5.2), which describe finite behaviour and can be reduced to process 0 by input, output, and τ actions on their free names.

Process abstraction $Test_{core}$ describes behaviour of a core functionality of component test, which implements a test script. The behaviour is finished after the test script is performed, so $Test_{core}$ is reduced to process 0. Analogously, process abstraction $test_{plug}$, which describes processing of first requests to stop and to detach a non-existent component before it can be replaced by a real component implementing a specific test script (e.g. component test), is performed only once and reduced to process 0. Process constants $Wire$ and $Dist$ describe behaviour of a connector of two interfaces and distribution of a start/stop request from a composite component among its subcomponents, respectively. Although they contain recursions and their behaviour can be infinite, they can be terminated instantly (e.g. when the connector is removed or the request has been already submitted to all of the subcomponents). In such case, process constants $Wire$ or $Dist$ can be reduced to process 0 (by means of an input action on name d or an output action on name r, respectively).

Agents `TE2comp` and `TEimpl` have deadlocks in processes that are reachable by 22 and 31 commitments, respectively. The deadlocks are related to the ability of process abstraction TE_{comp}, which describes behaviour of composite component testEnvironment, and of process abstraction TE, which describes behaviour of service `TestEnvironment`, to receive and to execute a test script. During the execution, behaviour of the component and the service is controlled by the test script (the component's subcomponent controller is waiting for an input on its interface `cDone`, see Section 2.2). If the test script is incompatible with its environment and can not be

[4] Nevertheless, these processes can come to a *live-lock* in their mutual co-operation. In such a case, the processes will communicate only between themselves and will periodically change, but as a whole system, they will not be responding to any external actions on their free names.

finished, the component and the service come to a deadlock.

In our approach, the deadlock-checking can be utilised to detect erroneous behaviour of individual services and components.

6.3. Bisimulation Checking

In π-calculus, *congruences* are equivalence relations[5] on π-calculus processes, which allows to formulate structural and behavioural equivalences between the processes. Two π-calculus processes express the same behaviour if they are *barbed congruent*, which means *bisimilar* in terms of labelled state transition systems, i.e. if no difference can be observed when they are put into an arbitrary π-calculus context and compared using the appropriate bisimulation game [20].

There are four important relations – namely an early strong bisimulation, a late strong bisimulation, an early weak bisimulation, and a late weak bisimulation. *Early and late bisimulations* differ in ways to treat input actions. *Strong and weak bisimulations* differ in ways to treat internal actions, the strong bisimulation treats internal τ-action and visible action equally while the weak bisimulation makes abstraction from the number of internal τ-actions (i.e. evolution of bisimilar systems is independent on their internal τ-actions).

The ABC allows to check strong and weak open bisimulation equivalences by means of commands "`eq`" and "`weq`". Moreover, in a case of of two agents that have the same free names, the bisimulation equivalences can be checked also by means of commands "`eqd`" and "`weqd`", which suppose the free names of the first agent are distinct from the free names of the second agent.

To demonstrate bisimulation checking in our case study, we check the equivalences of process $Test_{core}$ and its possible replacements. The process describes core behaviour of component `test` representing a test script (see Section 5). The bisimulation checking of behaviour of the original test script, which is supposed to be correct, and behaviour of its replacements, which may be wrong, can prevent the deadlock in agents `TE2comp` and `TEimpl`, as it has been described in Section 6.2.

In addition to agent `TestCore`, we define two agents with the same free names. The following definitions include original agent `TestCore` and new agents `TestCoreEquiv` and `TestCoreNonequiv`:

```
(*** TestCore ***)
agent TestCore = (\pexecuteWithID,rtInteract,rtResult)
   pexecuteWithID(id) . (^ret) 'rtInteract<ret>
   . ret(val) . 'rtResult<val,id> . 0
(*** TestCoreEquiv ***)
agent TestCoreEquiv = (\pexecuteWithID,rtInteract,rtResult)
   pexecuteWithID(id) . (^comm)
   ( (^ret) 'rtInteract<ret> . ret(val) . 'comm<val>
   . 0 | comm(res) . 'rtResult<res,id> . 0 )
(*** TestCoreNonequiv ***)
agent TestCoreNonequiv = (\pexecuteWithID,rtInteract,
                                                rtResult)
   pexecuteWithID(id) . (^ret) 'rtInteract<ret>
   . ret(val) . (^resid) 'rtResult<val,resid> . 0
```

Agents `TestCore` and `TestCoreEquiv` are not strongly open bisimilar, because agent `TestCoreEquiv` can perform an internal communication via name `comm`, that can not be performed by agent `TestCore`. However, these agents are weakly open bisimilar and according to ABC, a core relation[6] of their bisimulation contains 12 members.

The agents `TestCore` and `TestCoreNonquiv` are neither strongly open bisimilar nor weakly open bisimilar. The problem is at the end of processing, when agent `TestCore` sends via name `rtResult` name `id`, which has been previously received via name `pexecuteWithID`, while agent `TestCoreNonquiv` creates and sends a fresh name

[5] The *equivalences* are relations that are reflexive, symmetric, and transitive. The *congruences* ensure that if processes P and Q are in a relation of equivalence and process P is a subprocess (a component) of process R, then process R with substituted P for Q is in the relation of equivalence with the original process R (i.e. a substitution of equivalent components of processes does not break the equivalence of the processes).

[6] The core relation of bisimulation is a ternary relation between an agent, a set of distinctions, and an other agent, such that an union of its symmetric closure and the identity relation is a bisimulation [4].

`resid`, which differs from the original name `id`. The replacement of agent `TestCore`, which describes behaviour of component `test`, by agent `TestCoreNonequiv` leads to a deadlock (see the context of component `test` in Section 2.2).

6.4. Model Checking

Model checking is possible by means of the MWB, which uses π-μ-calculus [7], an extension of the μ-calculus[7], as a property specification language.

In MWB, we can check safety and liveness properties by means of μ and ν operators, respectively, as well as simply check the existence of specific reduction steps by means of modal operators $\Diamond$ and $\Box$. The following command verifies the ability of agent `System` to perform input actions on its free names `st` and `rl`:

```
check System(st,rl)<st>TT & <rl>TT
```

Agent `System` describes behaviour of the system from our case study (see process abstraction *System* in Section 4). The complete description of syntax and semantics of the π-μ-calculus in MWB can be found in [21].

7. Related Work and Discussion

Related works relevant to our subject can be divided into two groups, as formal approaches to describe service-oriented architectures (SOAs) and as formal approaches to describe component-based systems (CBSs). In this section, we outline current state of the art in both groups and discuss advantages and drawbacks of our approach, which intends to bridge the gap and to provide formal description of service-oriented architecture from choreography of services to individual components of underlying component-based systems.

In the first group, there are approaches mostly based on *Business Process Execution Language for Web Services* [2], such as [12], [15] or [22]. Those approaches focus on the web services, as a specific implementation of SOA, and provide formal description of choreography and orchestration based on business processes. The description ends up at the level of individual services implementing business processes and does not include underlying CBSs.

The second group consists of several component models[8] [14], such as Darwin/Tracta [11], Fractal [5] or SOFA 2.0 [6]. Those models usually focus only on pure CBSs without considering SOA at the higher level of abstraction. In some cases [13], the component models brings features of SOA into CBD, so that SOA becomes a specific case of a CBS. However, this solution mixes two different levels of abstraction (see Section 1).

Our approach is similar to the Reo coordination language [8], which is also based on π-calculus and able to describe both service in SOA and components in CBSs. In comparison with Reo and the above mentioned approaches (especially those in the second group), our approach describes services and components separately and with respect to their differences (i.e. services are not components and vice versa). We allow to go smoothly from services level to components level and describe behaviour of a whole system, services and components, as one π-calculus process. Moreover, we use standard polyadic π-calculus without any special extensions, which allows to utilise a wide range of existing tools for model-checking of π-calculus processes and formal verification of their properties.

However, our approach can have also drawbacks, e.g. complex description of behaviour of primitive components' control actions processing or insufficient visibility of a component-based system's structure during its evolution. After several dynamic reconfigurations and a corresponding sequence of reductions of the π-calculus process, it may be difficult to de-

[7] The (modal) μ-calculus is a temporal logic with a least fix-point operator μ and a greatest fix-point operator ν. It is used to specify properties of concurrent systems represented as labelled transition systems.

[8] I.e. meta-models of architectural entities, their properties, styles of their interconnections, and rules of evolution of the architecture of component-based systems.

termine a final configuration from the resulting π-calculus process, especially without knowledge of the exact sequence of reductions.

8. Conclusion and Future Work

We have demonstrated an approach to formal description of behaviour of service-oriented architecture on a case study of a testing environment of a railway interlocking control system. The approach is innovative, it captures behaviour of services as well as behaviour of underlying systems of components, yet it distinguishes these two levels. Future work is related to integration of the approach into modelling tools and automatic generation of the formal description.

Acknowledgements. This research has been supported by the Research Plan No. MSM 0021630528 "Security-Oriented Research in Information Technology".

References

[1] J. Amsden. Modeling SOA, parts I–V. *IBM developerWorks*, October 2007.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for Web Services, v1.1. Technical report, IBM, 2003.

[3] J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[4] S. Briais. *The ABC User's Guide*, May 2005.

[5] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium, February 2004.

[6] T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006*, pages 40–48, Seattle, USA, August 2006. IEEE Computer Society.

[7] M. Dam. Model checking mobile processes (full version). SICS Research Report R94:01, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden, 1994.

[8] N. K. Diakov and F. Arbab. Compositional construction of Web Services using Reo. In S. Bevinakoppa and J. Hu, editors, *Proc. of International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI 2004)*, pages 49–58. INSTICC Press, April 2004.

[9] R. Donini, S. Marrone, N. Mazzocca, A. Orazzo, D. Papa, and S. Venticinque. Testing complex safety-critical systems in SOA context. In *CISIS*, pages 87–93, Los Alamitos, CA, USA, December 2008. IEEE Computer Society.

[10] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, August 2005.

[11] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology and Medicine University of London, Department of Computing, January 1999.

[12] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets.

[13] P. Hnětynka and F. Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of CBSE 2006*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.

[14] K.-K. Lau and Z. Wang. A survey of software component models (second edition). Pre-print CSPP-38, School of Computer Science, The University of Manchester, Manchester M13 9PL, UK, May 2006.

[15] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:41–77, September 1992.

[17] M. Rychlý. A component model with support of mobile architectures and formal description. *e-Informatica Software Engineering Journal*, 3(1):9–25, October 2009.

[18] M. Rychlý. *Formal-based Component Model with Support of Mobile Architecture*. PhD thesis, Department of Information Systems, Faculty of Information Technology, Brno University of Technology, February 2010.

[19] M. Rychlý and P. Weiss. Modeling of service oriented architecture: From business process to service realisation. In *ENASE 2008 Third International Conference on Evaluation of Novel Approaches to Software Engineering Proceedings.*

Institute for Systems and Technologies of Information, Control and Communication, 2008.

[20] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New edition, October 2003.

[21] B. Victor. *The Mobility Workbench User's Guide*, polyadic version 3.122 edition, October 1995.

[22] M. Weidlich, G. Decker, and M. Weske. Efficient analysis of BPEL 2.0 processes using π-calculus. In *APSCC '07: Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference*, pages 266–274, Washington, DC, USA, 2007. IEEE Computer Society.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Defect Inflow Prediction in Large Software Projects

Miroslaw Staron*, Wilhelm Meding**
*Department of Applied IT, Chalmers | University of Gothenburg
**Ericsson SW Research, Ericsson AB
miroslaw.staron@ituniv.se, wilhelm.meding@ericsson.com

Abstract

Performance of software projects can be improved by providing predictions of various project characteristics. The predictions warn managers with information about potential problems and provide them with the possibility to prevent or avoid problems. Large software projects are characterized by a large number of factors that impact the project performance, which makes predicting project characteristics difficult. This paper presents methods for constructing prediction models of trends in defect inflow in large software projects based on a small number of variables. We refer to these models as short-term prediction models and long-term prediction models. The short-term prediction models are used to predict the number of defects discovered in the code up to three weeks in advance, while the long-term prediction models provide the possibility of predicting the defect inflow for the whole project. The initial evaluation of these methods in a large software project at Ericsson shows that the models are sufficiently accurate and easy to deploy.

1. Introduction

Large software projects have very different dynamics than small projects; the number of factors that affects the project is much larger than for small and medium software projects. Therefore, while constructing predictions for large software projects, there is a trade-off between the prediction accuracy and the effort required to collect the data necessary to predict (designated by the number and complexity of variables). The need to collect larger number of data is particularly important as the data may be distributed over time and across the globe. Ericsson is no exception in that. The current practices for constructing predictions in large software projects at Ericsson rely heavily on expert estimations, which are rather time consuming; in particular the experts use analogy based classification techniques while constructing the predictions for defect inflow – by identifying similarities and differences between projects, the experts construct the predictions.

In this paper we present a case study conducted at Ericsson which results in developing new methods for short-term and long-term defect inflow prediction. In our case (and at Ericsson), *defect inflow* is defined the number of defects reported as a result of executing test cases during the development project at a specific timeframe, often a week, a month or a year. The term *inflow* is used in the company to denote that the defects which are discovered have to be removed before the project concludes (i.e., the product is released) and therefore these defects constitute additional work inflow in the development project.

In this paper we present two methods for predicting defect inflow in large software projects in industry:

– short-term defect inflow prediction used to predict defect inflow for periods up to 3 weeks on a weekly basis, and
– long-term defect inflow prediction for the entire project lifecycle on a monthly basis.

For each project, the long-term prediction model provides means of planning projects and allocating resources by taking into consideration expected number of defect detected which have to be removed before the release. The short-term prediction model provides the means of immediate monitoring of the defect inflow status in the project. The methods complement each other as the predictions from the short-term model need to be interpreted in the context of the predictions from the long-term model.

In this paper we also present an evaluation of these two methods in a large software project at Ericsson. The results of the evaluation show that both the short-term and long-term prediction models developed using our methods increase the prediction accuracy in comparison to the existing practices.

The structure of the paper is as follows. Section 2 presents the most relevant work for this paper. Section 3 introduces the context of the case study, in particular the organization of the large projects for which the methods are constructed. Section 4 describes the short-term prediction method, while Section 5 presents the long-term prediction method. Section 6 presents the process of evaluation of the prediction models and Section 7 presents the results from the evaluation and Section 8 validity evaluation of our study. Finally, Section 9 contains the conclusions.

2. Related Work

The most related work to our long-term prediction method is the work of Amasaki [2] who also aims at creating defect inflow profiles and trends. Their work is focused on using trends from development project to predict post-release defect inflow, which is different from our work. Our methods are intended to predict the defect inflow trends in the development project (i.e., when the software product has not been released yet) with the aim to support project management rather than maintenance of the product.

Our long-term defect inflow prediction model is similar to the the HyDEEP method [9] by Klas et al. The HyDEEP method aims to support product quality assurance similarly to our work. The HyDEEP method requires establishing a baseline for effectiveness of defect removal process which requires additional effort from the quality managers (compared to our method). The mean relative error of the predictions created by HyDEEP can be up to 29.6% which makes it a viable alternative to our long-term defect inflow prediction. In our future work we intend to compare the HyDEEP method to our long-term prediction in our industrial context to compare the complexity and ease-of-use of these two methods.

When developing the long-term defect inflow prediction model we used a similar approach to Goel [7]. Goel's process advocates for using one of the predefined defect inflow prediction model – e.g. Goel–Okumoto Nonhomogeneous Poission Model, whereas we advocate for using linear regression methods. The models considered by Goel usually consider non-linear dependencies between defect occurrences and between testing and defect discovery. Based on our discussion with experts at the company the dependencies in our case are linear cause-effect relationships.

Our research on long-term defect inflow prediction models is similar to the research on reliability theory in software engineering w.r.t the fact that we are interested in the defect inflow profile and not defect density during development. One of the most well-known models used in the reliability theory is the Rayleigh model [10], which describes the defect arrival rates for software projects after the release. It was the first model we attempted to adjust and apply before we created the method presented in this paper without much success. We use this model in the evaluation to show why this model was not appropriate for our case. Our work is substantially different from the models which predict the defect inflow after release for the following reasons:

– we can assume that when the development project is concluded, the defects reported are taken care of during the maintenance project (which can use the reliability theory to predict the defect inflow),
– changes in the trends of the defect inflow are caused by the project milestones and do not depend on the time after the release.

These underlying differences from the reliability theory underpin our research and make us have a different approach than the reliability theory.

Li et al. [11] evaluated empirically the ways of predicting field defect rates using Theil forecasting statistics. The results of their work influenced the design of our study on long-term defect inflow prediction, although we see their assumption of defects being reported after the release to be the main difference in contexts from our research. The Software Reliability Growth Model [13], which is used in their paper, seems not to be applicable to the development projects done in iterative way, since it assumes that no new functionality is developed when the defects are reported; this is not the case of the development projects.

In our future work we consider extending our prediction methods by using the same principles of Constructive Quality Modeling for Defect Density Prediction (COQUALMO) [4] and its recent extension – Dynamic COQUALMO [8]. In particular we intend to consider defect introduction and removal as two separate processes and use the efficiency of defect removal (from [9]) as one of the variables in our prediction models in the short-term predictions.

In our short-term models we consider the defect inflow to be the function of characteristics of work packages (e.g. the accumulated number of components reaching a particular milestone) and not directly the characteristics of the affected components (e.g. size or complexity). Using the characteristics of components as the sole predictors would provide us with a possibility to predict the defect density of the component and present this data on a monthly/weekly basis (based on when the component will be put under testing). Such an approach would be an extension of the current work on defect density, e.g., [3, 16, 12, 1]. In our case, nevertheless, this approach seems not feasible, because the information about how the components are to be affected by the project is not available at the time of developing predictions; in particular the change of size and complexity is not available. For short-term predictions, the data on size and complexity of components was not available on a weekly basis simply because measuring the size and complexity change is not meaningful for particular weeks; the measurements of component characteristics are done according to project plans – e.g. builds – and not on a weekly basis (i.e., not according to calendar time). In our further work we intend to evaluate if it is feasible to re-configure this data and use it as an auxiliary prediction method.

Our research on short-term defect inflow can be extended by using the research of Ostrand et al. [17] on predicting the location of defects in software components in large software systems. Predicting the location of defects can be applied as the next step after the long-term defect inflow predictions are in place, to guide the project managers into channelling testing efforts into components (or work packages which affect these components) which are historically responsible for the largest amount of defects in the system.

When developing the short-term defect inflow prediction models we considered using capture-recapture techniques [18] for estimating the number of defects in the product to assess the viability of our predictions. However, we decided to prioritize the simplicity of data collection since using capture-recapture data would require additional effort from the testers when reporting discovered defects and a more thorough statistics of the data from the defects database.

3. Context

The context of the case study is Ericsson and one of its large projects, which is developing one of the releases of a network product. In the course of development of the methods we used the pre-

vious releases of the product and we applied the methods on the new release of the product. The product has already had several releases and it can be considered as a mature one. Choosing late project releases decreases the risk of using data biased with immaturity in the organization, as it has already been shown by [21, 22] in a similar context at the same company.

As a new practice at Ericsson, the large software projects are structured into a set of work packages which are defined during the project planning phase. In the projects which we studied, the temporal aspects of defect discovery were more important than the total number of defects for the products. This was dictated by the fact that stakeholders for this particular work were project managers and at the project management level, the defect inflow is a measure of extra effort in the project (as the discovered defects have to be removed from the product before the release). The number of defects discovered at a particular point in time seems to be a function of the number of work packages reaching the testing phase. Complexity and size characteristics of the product do not have a direct impact on the number of defects discovered in a particular time frame, but the total number of defects discovered in the product.

In the existing prediction work on defect density [3, 16, 12, 14] it is usually the case that a component is developed by a single work package (or even the project, depending on the size of the component and project). In the case of Ericsson, work packages are related to the new features being developed and seldom result in creating completely new subsystems or single system components. The division of project into work packages is based on customer requirements, while the division of system into sub-systems and components is based on such elements as architectural design and the architecture of the underlying hardware (hardware/environment constraints). Each work package develops (or make changes to) components for each new large project, which makes it hard to develop a unified defect inflow prediction model using measurements at the component level. During the whole product life cycle (which spans over more than one large project – also referred to as release from the perspective of the product) the division of projects into work packages changes to a large extent (as the requirements are different for every release). Therefore using work package characteristics (which are based on distributing functionality) makes the method for long-term predictions generalizable to other projects at Ericsson.

Furthermore, from the perspective of project management, the defect inflow is also a function over the status of the project, i.e., where in the lifecycle the project is. This information is conveyed by the work package completion status. This is the assumption that we use in constructing the defect inflow predictions – that the defect inflow rate is dependent on where in the lifecycle the project is. During the project lifecycle there are 3 major milestones which are important for the long-term prediction method:

- Md – design ready milestone, which defines the point when all work-packages have finished designing. The Md milestone is used as a reference point when comparing the baseline and predicted projects,
- Mt – test ready milestone, which defines the point when all work-packages have finished their tests,
- Mf – product finished, which defines the point when the development project concludes and the product is released (the maintenance organization takes over the responsibility for the released version of the product – for that period of time the reliability theory can be used to predict the defect inflow).

Predicting the defect inflow in the large software projects at Ericsson is an important task for project planning (long-term predictions), project monitoring, and early warning mechanism (short-term predictions). Ericsson's quality managers created the predictions manually using expert opinions and analogy based techniques. The process of creating the predictions was time consuming and resulted usually in creating predictions for 2 months and interpolating the remaining months using straight lines.

The short term predictions were not widely used in the company due to the fact that they were effort intensive. The new organization of software projects provides a unique opportunity to create more accurate and less time-consuming prediction models using statistical regression techniques.

4. Short-Term Defect Inflow Prediction

In this section we present the process of developing the short-term prediction model, introduce methods used for developing the model, present the development of the model, and finally provide a short example based on a case from Ericsson.

4.1. Process

The process of creating predictions in our case started by using a baseline project data to develop the prediction models, which resulted in a set of candidate models. The applicability of these candidate models for predictions was assessed by examining the R^2 model-fit coefficient [23]. The candidate model which had the highest R^2 was chosen for further development. This model was used on a new set of data from one of the current projects to check if it was applicable for predictions. The check was done by calculating *Mean Magnitude of Relative Error* or MMRE [5]. If the MMRE was sufficiently low (below 30%, which was arbitrarily chosen by Ericsson), then the model was used for predictions. If the MMRE was higher than 30% then the "second-best" candidate model was used with the new set of data and a new MMRE was calculated.

4.2. Methods

When developing the models we used a number of statistical methods. We decided that the prediction model would be in form of a linear equation – an outcome of *multivariate linear regression* method [23]. The choice of linear regression was dictated by the dependencies between measures (*predictor variables*) at Ericsson. Based on discussions with experts in the company we could not identify polynomial or exponential dependencies in short-term predictions which dictated the use of linear model. To construct the model we used the previous release of the product, which we found to be similar in size, complexity, and maturity of project teams to the new projects in the company.

In order to avoid problems with co-linearity within our data set we used *Pricipal Component Analysis* or PCA [6]. Principal component analysis was performed prior to multivariate linear regression and was used to identify variables in the data set which can explain most variability in the data set. We did not use the principal component since our goal was to use as little data as possible and still be able to have accurate predictions.

To create the multivariate linear regression equation we used the method of Least Squares [23] for fitting coefficients in the equation and we used the model-fit coefficient R^2 [23] when evaluating whether the model can be used for predictions. When we evaluated the model on a new data set from the current project we used MMRE [5] to check how well the predictions fit the actual values of defect inflow. The evaluation methods are described in more detail in Section 7.

4.3. Model Development

4.3.1. Assumptions and Application

The short-term defect inflow prediction model was based on the following assumption: we used data from past weeks to describe the defect inflow for the current week. In other words, the dependent variable in the model was the defect inflow for the current week (which has already been known), while the independent variables were the defect inflow from previous weeks and planned/actual milestone completion status for the current week and the previous weeks.

This assumption meant that once we had the regression model describing the defect inflow for a particular week using data from the past

weeks, we could use this model to predict the defect inflow for the coming weeks by substituting data for past weeks with the data from the current week. As the predictor variables we used the defect inflow for the current project (for the time up to till the current week), and milestone completion status.

The short-term prediction model allowed predicting the defect inflow for 1–5 weeks in advance, for example: if we found that our predictor variables were number of defect inflow from 5 weeks before and the accumulated planned Md completion, using the data for the current week, we could predict what the defect inflow will be in 5 weeks. However, through simulations we found that the predictions for future weeks 4 and 5 had low prediction accuracy and therefore they are not discussed in this paper.

4.3.2. Choice of Predictor Variables

The choice of predictor variables was dictated by the availability of data and our goal – to make predictions based on the data that already existed in the organization and was easy to obtain. As discussed in Section 3, we could use milestone completion and test progress to predict defect inflow (since these were relevant variables and they influenced the defect inflow). The source of the defect inflow: testing was performed before both the design ready milestone (Md) and test ready milestone (Mt). We discovered that we needed to distinguish between Md and Mt phases as Md is only used before the Md date for the project and Mt completion status is used after the Md date.

In our case study we took into consideration the following predictor variables:

- Number of planned Md completions (prefixed with Mdp), and accumulated number of planned Md completions (prefixed with AMdp) – accumulated number of completions is the number of completions from the beginning of the project until the current week – for
  - The predicted week (Mdp_0, $AMdp_0$),
  - 1 week before (Mdp_1, $AMdp_1$), 2 weeks before (Mdp_2, $AMdp_2$), ..., 5 weeks before (Mdp_5, $AMdp_5$) the predicted week,
- Number of actual Md completions (Mda) and accumulated number of actual Md completions (AMda) for
  - 1 week before (Mda_1, $AMda_1$), 2 weeks before (Mda_2, $AMda_2$), ..., 5 weeks before (Mda_5, $AMda_5$) the predicted week;
- Number of planned Mt completions (Mtp) and accumulated number of planned Mt completions (AMtp) for
  - The predicted week (Mtp_0, $AMtp_0$),
  - 1 week before (Mtp_1, $AMtp_1$), 2 weeks before (Mtp_2, $AMtp_2$), ..., 5 weeks before (Mtp_5, $AMtp_5$) the predicted week;
- Number of actual Mt completions (Mta) and accumulated number of actual Mt completions (AMta) for
  - 1 weeks before (Mta_1, $AMta_1$), 2 weeks before (Mta_2, $AMta_2$), ..., 5 weeks before (Mta_5, $AMta_5$) the predicted week;
- Number of reported defects (defect inflow, Di) for
  - 1 week before (Di_1), 2 weeks before (Di_2), ..., 5 weeks before (Di_5) the predicted week.

To avoid problems with multi-collinearity we used the variables that were not correlated and we chose the data from the project plan which can always be obtained early in the project. A representative scatter plot for relationship between two of these variables is presented in Figure 1.

While constructing the defect inflow prediction models we deliberately did not include the data for product size/complexity as this data was not related to project planning for the following reasons:

- the software components were not assigned on a one-to-one basis to work packages and the milestones characterize work packages, not the components,
- the data on source code size was collected for milestones in the project (as it does not make sense to collect them on a weekly basis) – which means that for the whole project we could use few data points for size,
- the organization was concerned with project planning and monitoring, and not source code characteristics (e.g., size is only an in-

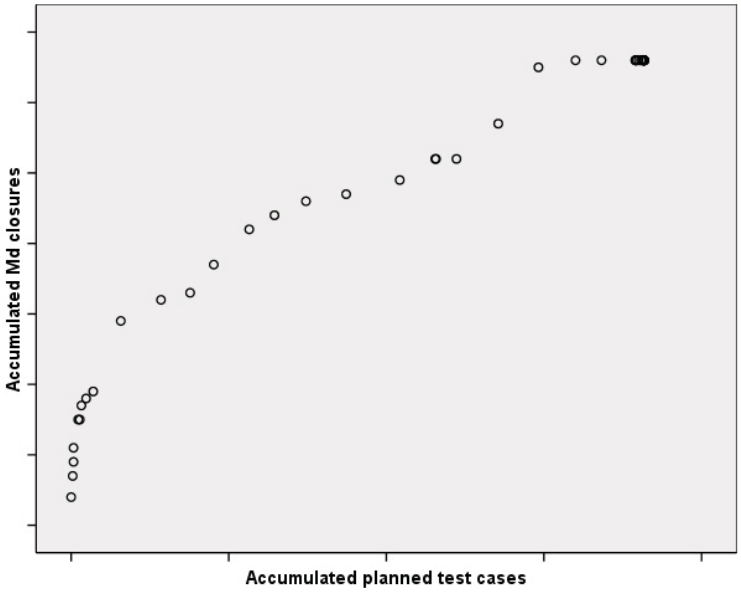


Figure 1. Relationship between accumulated planned test cases to be executed and Accumulated Md closures, Spearman's correlation coefficient: 0.96

put to the planning, but is not monitored in the projects).

Neither did we decide to use data about Md/Mt completion status as the accumulated numbers of test cases planned and executed were highly correlated with the Md/Mt completion status (Spearman's correlation coefficients [23] between 0.96 and 0.99 significant at the 0.01 significance level).

In our search for the predictor variables we used a large number of approaches and experimented with more variables than the above ones. However adding more variables did not increase the accuracy of the model significantly. For instance using all relevant variables (ca. 30) in one of the equations increased the accuracy by only 1%, but the additional effort for data collection increased significantly.

4.3.3. Reducing Number of Predictor Variables

Before constructing the regression model over the candidate variables, Principal Component Analysis was used to reduce the number of variables and thus identifying the strongest predictors. We experimented with the initial set of variables (the input to PCA) in order to achieve the best possible percentage of explaining the variability at the minimal set of measurements. PCA analysis identified 4–7 principal components (depending on the prediction period: 1, 2, 3, 4, or 5 weeks in advance). The scatter plot for the main principal components and the defect inflow is presented in Figure 2. Due to the confidentiality agreements with our industrial partner, the values on the Y-axis are not provided.

We used PCA to identify the key components and we used variables which constituted these components for the prediction models. We re-calculated the loadings in the components so that we could present the equations using the original variables and not the components (as this was one of our requirements while deploying the model in the company – to use the original names of measurements, not the names of the components).

4.4. Result: Short-Term Defect Inflow Prediction Model

The principal components before Md seemed to be linearly correlated to the defect inflow, which made the linear regression a viable technique for building the prediction model in this case. For the principal components after Md, the compo-

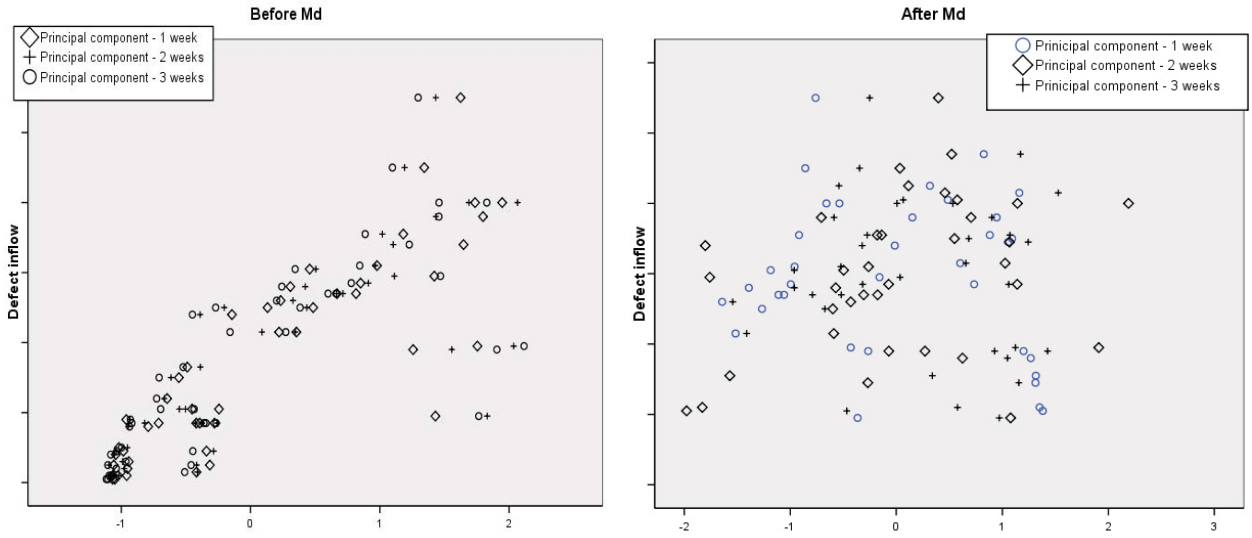


Figure 2. Scatter plot for the main principal components and defect inflow for the models before Md and after Md. The X-axes show the values of the components

nents did not show a strong correlation with the defect inflow, which affects the fitness of the regression models and the prediction accuracy. We checked whether the principal components expose logarithmic and polynomial relationship to the defect inflow, but the results showed almost a complete lack of relationship for the logarithmic curve and large scatter for the polynomial curve. This supported the claims from Ericsson experts that the relationships are linear and not polynomial or exponential. The variability of the data set explained by the principal components is presented in the last column in Table 1.

The equations used to predict the short-term defect inflow are presented together with the R^2 coefficient for the regression model and the variability explained by the component which contained the variables used in the equation. The variables used in the equations are subsets of variables presented earlier in this section.

4.5. Example

As an example, let us predict the defect inflow for a particular week in an example project. Let us assume that we are in week 10 of the project, and it is before the design ready milestone (Md). The data for that particular week is presented in Table 2. We predict week 11, so the values for 1 week before the predicted week are the values for week 10 (the current week), the values for 2 weeks before the predicted week are the values for week 9, etc. We substitute the appropriate numbers for the equations presented in Table 1 thus obtaining the predictions for week 11. By using the data from week 10 as the data for 2 weeks before, data from week 9 as the data for 3 weeks before, we can create predictions for week 12 – for example AMdp2 in equation is AMdp1 from Table 2 (because the data shows the values relative to week 10, not week 12. The value of the defect inflow for week 13 is obtained in the same way.

The results for the short-term predictions are presented in Figure 3.

The figure shows that given the current circumstances of the project (i.e., the number of defects reported in the current week and the status of the planned and accumulated numbers of work packages reaching the Md milestone) week 13 seems to be the week when the project manager should pay more attention to as the number of defects discovered is going to be rather high. The potential action of the project manager is to prepare more development resources for that week to repair the defects discovered.

Table 1. Defect inflow prediction models (short-term)

Before/after Md (design ready milestone)	Period	Equation	R2	Variability explained by component
Before Md	1 week	$D = 1.499 + 0.584 * AMdp_0 + 0.650 * Di_1 - 1.285 * AMdp_1 + 1.102 * AMda_1$	0.86	93.84%
Before Md	2 weeks	$D = 2.639 + 1.173 * AMdp_0 - 2.029 * AMdp_2 + 1.724 * AMda_2 + 0.461 * Di_2 - 0.366 * Di_3$	0.82	91.43%
Before Md	3 weeks	$D = 4.187 - 0.357 * Di_3 + 1.928 * AMdp_5 - 1.192 * AMdp_0$	0.62	90.92%
After Md	1 week	$D = 39.155 + 0.470 * Di_1 + 1.290 * AMtp_0 - 1.185 * AMtp_1 - 1.214 * AMta_1 + 0.039 * AMtp_2 - 0.044 * AMta_2 + 1.297 * AMtp_3 - 0.517 * AMta_3 + 0.003 * AMtp_4 + 0.107 * AMta_4 + 0.148 * Di_2 - 0.237 * Di_3 - 0.194 * Di_4 + 0.180 * Di_5$	0.67	65.74%
After Md	2 weeks	$D = 53.669 + 1.419 * AMtp_0 - 0.682 * AMtp_1 + 0.099 * AMtp_2 - 1.844 * AMta_2 + 0.429 * AMtp_3 - 0.768 * AMta_3 + 0.646 * AMtp_4 + 0.446 * AMta_4 + 0.306 * Di_2 - 0.265 * Di_3$	0.55	78.69%
After Md	3 weeks	$D = 24.82 + 0.47 * Di_3 - 0.351 * AMtp_2 + 0.308 * Di_5$	0.62	59.09%

Table 2. Data for week 10 of the project

Variable	$AMdp_0 - week11$	$AMdp_0 - week12$	$AMdp_0 - week13$	$AMdp_1$	$AMdp_2$	$AMdp_5$	$AMda_1$	$AMda_2$	Di_1	Di_2	Di_3
Value	5	5	5	13	14	20	12	12	4	5	5

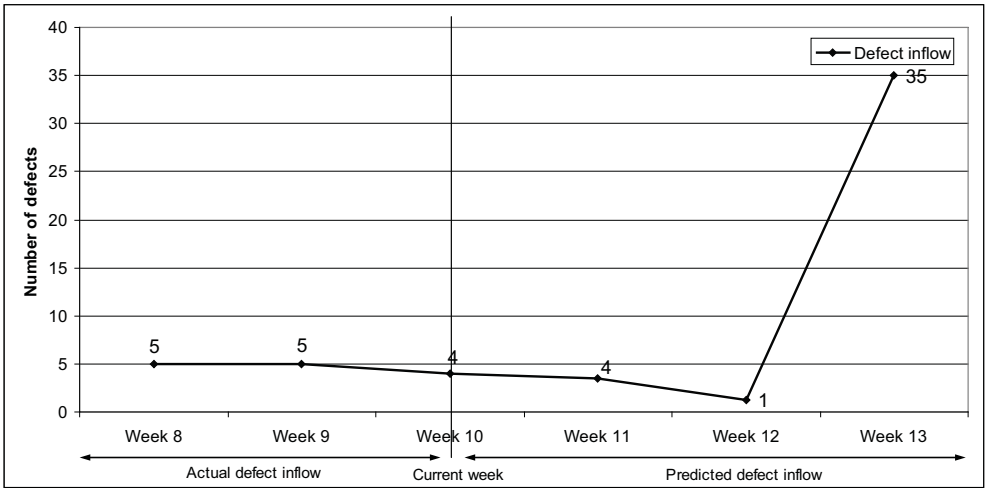


Figure 3. Short-term defect inflow prediction for week 10

5. Long-Term Defect Inflow Prediction

While the rationale behind the short-term prediction was to predict the status of the project, the rationale for the long-term prediction was to create a monthly trend of defect inflows in the project for the whole duration of the project. The process for creating the model was the same as for the short-term prediction, but the methods and results were different.

In the case of short-term models it was the equations which were the resulting model. In the case of long-term prediction it was the method which was the result. This method consists of the following steps:

1. Identify a baseline project,
2. Partition the defect inflow curve of the baseline project,
3. Create regression models for each part of the curve,
4. Calculate scaling factor for the new project,
5. Plot the defect inflow curve for the new project.

On the contrary to the short-term defect inflow prediction, the long-term prediction used a single predictor variable – project progress – unlike several variables in the short-term prediction models.

5.1. Long-Term Defect Inflow Prediction Method Development

5.1.1. Methods

For creating the regression models we used the polynomial regression method and we used only one variable – the project progress – as the predictor variable. The regression use the Least Squared Estimators similar to the multivariate linear regression used for short-term predictions.

For creating the scaling factors we used an average ratio between the number of defects reported in the new project compared to the baseline project. The calculation of this factor Sc was done using the following formula

$$Sc = \frac{1}{n}\sum_{i=1}^{n}\frac{p_i - a_i}{p_i},$$

where: n – the number of months for which we have the actual data, p_i – the value obtained by from the equation before scaling, a_i – the value of the actual data of defect inflow for this month. The rationale behind this formula was that it was an average relative difference between the predicted defect inflow and the actual data from the predicted project.

The scaling factor Sc was calculated for the curve for which there was some initial defect inflow data available. If the data were not available the scaling factor can be expert estimations (e.g. before the project start).

The method for calculating the scaling constant $Sc1$ for the curves which did not start at the beginning of the project was

$$Sc1 = \frac{pred_{ActualProject}}{pred_{BaselineProject}},$$

where: $pred_{Actualproject}$ – the predicted defect inflow for the current project for the last month for which the 1st equation (curve) can be used, and $pred_{Baselineproject}$ – the predicted defect inflow for the equation describing the second curve.

5.1.2. Assumptions and Application

The assumption for the long-term defect inflow prediction in the software project was that there existed a number of projects which could be used as a baseline. With that respect the long-term defect inflow prediction was similar to the analogy-based estimations.

The long-term prediction method should be used at the beginning of new projects in order to estimate how many defects can be discovered and when during the project. This information is particularly important for project planning and monitoring and the stakeholders for these predictions are project managers in large software projects. The long-term defect inflow prediction is usually not applicable for small projects since these usually different significantly from each

other, even if they are executed by the same organization and on the same product.

Our long-term defect inflow prediction model was designed to work best for projects already in progress as it used the actual reported defect inflow from the projects to adjust the prediction models and thus increase their accuracy. Using expert estimations at the beginning of the project should be replaced as soon as real data is available in the new project.

5.2. Result: Method for Constructing Long-Term Defect Inflow Prediction Models

It should be noted that the resulting long-term prediction model should be adjusted by the owner of the prediction in order to increase its accuracy. In particular, the defect inflow curve needs to be maintained once a month so that the predicted defect inflow is as accurate as possible.

5.2.1. Identify Similar Projects

The identification of the most similar project needs to be done by experts, e.g., the experts involved in creating the predictions. The factors that should be taken into account while identifying the most common projects are:

1. Estimated size of the project, measured as number of person-hours in the project.
2. Number of "heavy" features in the project – i.e., features which are of high complexity.
3. Complexity of the complete project – i.e., including integration complexity.
4. Time span of the project.

The most important factor is the estimated complexity. A rule of thumb used by experts is that for a project which is twice as big as a reference project, the number of defect inflow in each month would be around 75% more than in the reference project.

The time span is important only in the case when the projects are significantly longer, otherwise the method compensates for that by using the time relative to the Md milestone. Hence, similar to the short-term defect inflow prediction the Md milestone datse is an important reference point. In case the time span of the project is significantly longer, the scaling factor (described later in this section) needs to be obtained through expert estimates and not using the method described in this paper.

5.2.2. Partition the Defect Inflow Curve of the Baseline Project

In order to identify curves, identify the points where the curves change shape. These changes are important to identify otherwise we assume that the fitted equations will under-/over- predict the values of the peaks in the defect inflow. The peaks, however, are the most crucial elements to predict since project management is interested in the information how many defect might come in the peak time.

It should be noted that projects are usually independent from the calendar time – which means that the prediction model should be done according to the project milestones – e.g. Md. However, changes in the time scale should be done after the equations are built and when the models are to be applied for the current project.

5.2.3. Create Regression Models for Each Part of the Curve

The next step is to create equations describing the shape of the curves in this chart using regression methods. The equations for the curves can be identified using curve estimations methods in statistical packages. These curve estimations use the standard regression techniques – e.g. least square estimators.

In order to create the equations describing each curve (identified in the previous section) we need to:

- Use separate equation for each curve,
- Start with a straight line, and
- Change the curve type to polynomial with order set to 2, 3 or 4. The order depends on

the R^2 – as a rule of thumb, it should be as small as possible. The higher the degree of the polynomial, the higher the risk of errors in predictions when the time-scale of the predicted project is different than the time-plan of the baseline project.

These equations "re-create" the defect inflow for the baseline project using mathematical equations, which allows us to adapt the defect inflow trends for different time scales.

5.2.4. Calculate Scaling Factor for the New Project

Up to this point, the method constructs a set of equations which describe the trend of defect inflow in the baseline project. In order to predict the defect inflow in the future projects, the equations need to be scaled (moved up or down the y-axis) to reflect the actual values so far of the predicted project. The goal of this step is to have a prediction model which is in form of a set of equations which are scaled according to certain criteria. For the different parts of the curves identified so far we use different scaling factors. For the first curve, in our case there is a single criterion: the predicted defect inflow should fit the actual defect inflow from the predicted project. The outcome of the scaling is the scaling factor Sc, which is used in the following way

$$D_m(month) = Sc * y(month),$$

where: $D_m(month)$ – defect inflow for a specific month, $y(month)$ – the predicted defect inflow for the month calculated from the equations describing the baseline project.

As mentioned in the methods section there are two different ways of creating the scaling factor, which are used (i) at the beginning of the project (when no defects have been reported), and (ii) during the project when some defects have been reported. In the first case (i), the scaling needs to be done using expert opinion – i.e., the expert has to provide the ratio of complexity between the predicted and this ratio becomes the scaling factor Sc.

In the latter case (ii), the scaling can be done by "fitting" the predictions to the existing trend in defect inflow for the predicted project. Using the actual data means that we do not rely on subjective estimates but we actually try to answer the question: "How will the defect inflow look like in the predicted project if we continue with the current trend of defect inflow?" Furthermore, the predictions of defect inflow become more important once the project progresses – i.e., since the defect inflow is not expected to be large at the beginning of the project. This fitting can be done in the following ways: (i) Calculating the average relative difference, or (ii) Using non-linear regression. The first one (i) works well for the projects which has similar life span (i.e., differences in life spans should not be more than 2 months). The second (ii) is more robust and does not have this limitation.

5.3. Example: Long-Term Defect Inflow Prediction Model

In this section we present how we constructed a prediction model for one of the projects at Ericsson. In our study in one of the product the trends in the defect inflow for some of the releases are presented in Figure 4. The trends are similar, but not the same, which requires more advanced methods for predicting the defect inflow than only the analogy based estimations. Due to the confidentiality agreement with our industrial partner, we present the data scaled to the largest defect inflow in each project and we present only the subset of months for the project.

The figure shows that the trends in the defect inflow are rather stable over releases (although they differ in the values, which cannot be shown in the figure due to confidentiality agreements). They are presented in a relative time scale with the common point of Md milestone; the milestone when all work packages have finished designing.

An important observation is that all three projects have a similar percentage of defects in-

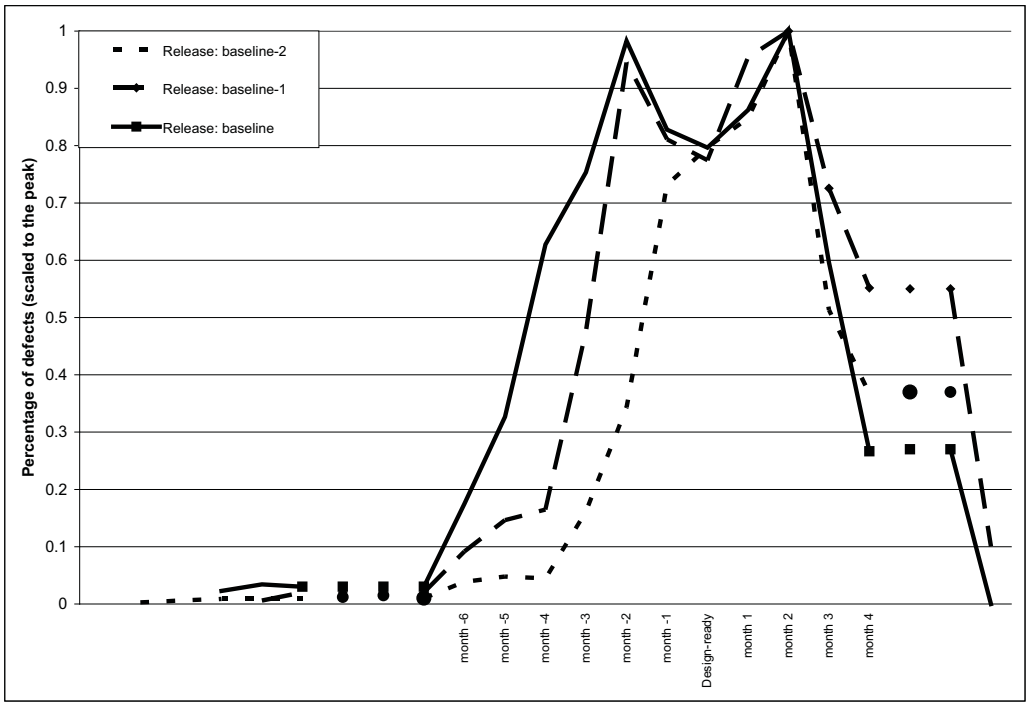


Figure 4. Defect inflow trends for previous projects scaled towards design ready milestone. The dots represent removing a number of months since we cannot show the complete time frame for project

flow at the Md milestone month. Since this is the case, we use the Md milestone month as a reference point in constructing the long-term predictions.

In order to identify the baseline project we asked experts who work with these baseline projects. They identified previous release of the same product as the best baseline ("Release: baseline" in Fig. 4).

In the case of this baseline project we identified the following curves:

1. months 1–5,
2. months 5–9,
3. months 9–11.

The developed trend line for months 1–5 is presented in Figure 5. The values of the defect inflow are normalized, due to the confidentiality agreement with our industrial partner. The curve equation is displayed in the chart, where x denotes the month. The scaling factor was calculated to be 1.23 in month number 3, which meant that the new project produced ca. 23% more defects thanAcknowledgments the previous project. We validated that with the quality managers for that project who confirmed that this number reflected their expert opinion.

6. Evaluation of Defect Inflow Models in the Context of Ericsson

6.1. Design of Evaluation

When developing the prediction models we used the model fit coefficient (R^2) to observe whether models are accurate w.r.t. the past projects used to build the models. In this section we describe how we used the data from new (current) projects to check whether the models accurately predict defect inflows in new projects. We evaluated two aspects: the ability to predict the correct value and whether predictions over a period of time (e.g., predictions 3, 2, and 1 weeks in advance) predict the same value (*stability*). Unstable models change rapidly over time which makes them less trustworthy – how can we trust a prediction model that will change a lot in the coming weeks/months?

We used the Magnitude of Relative Error (MRE) metric to measure how accurate the predictions are. MRE was defined in [5] as

$$MRE = \frac{|a_i - p_i|}{a_i},$$

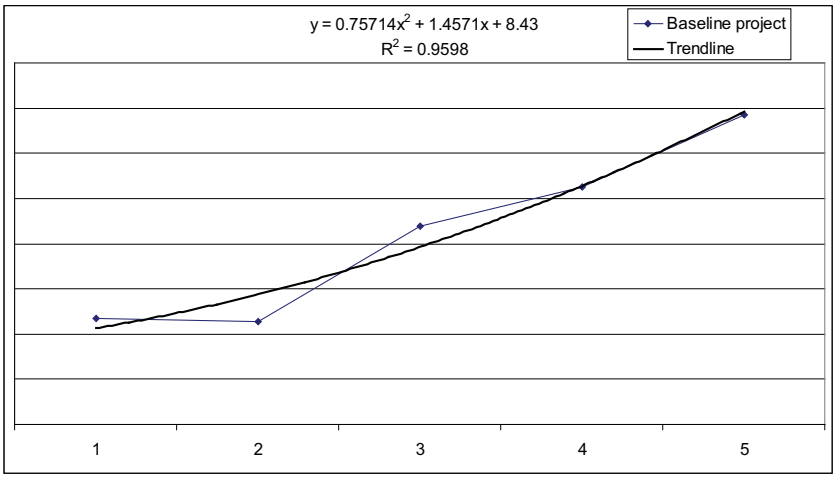


Figure 5. Equation for the curve for months 1–11

where a_i means the actual value for the defect inflow for i-th week (short-term predictions) or month (long-term predictions); p_i denotes the predicted value of defect inflow for the i-th week or month. In the evaluation we used both the distribution of MRE and mean MRE (MMRE). The best models were expected to have the lowest value of MMRE – i.e., the mis-predictions of models are small.

For evaluating the stability we used our own measure *mean in-stability (MiST)* as a metric for comparison, which we defined as

$$MiST = \frac{1}{n}\sum_{i=1}^{n}\frac{|m0_i - mj_i|}{m0_i},$$

where: n – the number of months used in the prediction, $m0_i$ – the predicted defect inflow for the i-th month created before the project (0th month), mj_i – the predicted defect inflow for the i-th month created in the j-th month.

In evaluation of the prediction accuracy we compared models developed in our research (presented in Table 1) and "average" models, i.e., predicting using a simple average amount of defect inflow in a baseline project (or the average number of defects in the current project – up to the week for which the prediction was made), and the moving averages. The rationale behind the average models was that if we did not know how to predict the number of defect inflow in a particular week, we could take the average number of defects for all weeks as an estimator; alternatively we could also use the median (i.e., the most common value of the defect inflow). Thus, in the evaluation (Figure 6), we used the following models:

- Average number of defect inflow from the baseline project,
- Average number of defect inflow from the actual project until the week of prediction,
- Moving average (2 weeks) of the number of defect inflow from the current project (i.e., the predicted value of the defect inflow is the average of the defect inflow from previous 2 weeks),
- Moving average (3 weeks) of the number of defect inflow from the current project (i.e., the predicted value of the defect inflow is the average of the defect inflow from previous 3 weeks),
- Value of the mode of the defect inflow from the baseline project (to some extend this is the use of analogy based estimation),
- Value of the mode of the defect inflow from the current project,
- Expert estimations for 1, 2, and 3 weeks.

In order to evaluate the long-term prediction models developed using this method, we compared the prediction model developed using our method to the following prediction models:

- Linear, quadratic, cubic, 4th degree, 5th degree, and exponential curve depicting the trend in defect inflow,
- Rayleigh model,

– Expert estimations, which are based on the predictions done for previous projects, prior to our research.

The above models were chosen since they require a similar amount of effort to create compared to our models. We have deliberately excluded methods like Bayesian Belief Networks [15] as their construction requires significantly more effort than the construction of the simple prediction models using our method.

6.2. Results of Evaluation

6.2.1. Short-Term Predictions

In this section we show how the models worked in a new project at the company. The new project which we chosen for the evaluation is the next release of the same product, while the baseline project was the previous release of the same product.

The values for the MMRE for the reference prediction models and the short-term prediction models are presented in Figure 6. Figure 6 indicates that the best model is the moving average for 2 weeks, which is one of the simplest models to construct. Our prediction models have larger mis-predictions, which are caused by the fact that the predictions after Md are based on the data which was weakly correlated with the defect inflow. Despite a low value of MMRE for predictions using moving averages, there is a disadvantage of these two models. Using moving averages shows trends in the defect inflow for more than one week, which are rather stable (the moving average are partially used by experts in estimating the defect inflow). However, the moving averages do not allow predicting peaks (as the peak shown in Figure 3 for week 13).

The worst prediction models are the models using modes from current and baseline projects; these two models mis-predict the defect inflow in most cases by more than 100%.The predictions made by the expert had the most common mis-predictions of 75%-90%. The predictions created using moving averages can result in less accurate models (since they have a larger percentage of mis-predictions above 90% than 'our' models). From the experiments with the historical data we found that the prediction models developed in this paper had a tendency of over-predicting (i.e., predicting values that were larger than the actual values), in particular indicating the potential "red-alerts" for the projects – i.e., showing that there will be a high raise in the defect inflow in the project. Although this might be a problem from the statistical perspective (low accuracy), this provides a means for project managers to get early warnings of potential problems so that they can have a time for reacting to some extent. This, however, cannot be verified on the historical data and we are currently in the process of evaluating the models in other large projects at Ericsson. The interview with the expert provided the predicted values for 10 different weeks. The expert was asked to make the predictions for 10 different weeks in the same way as he does the predictions when they are needed (using the information only up to the predicted week), although making short-term predictions is not done very often; the experts are focused on long-term predictions for the whole project. The results show that the methods presented in this paper are not worse than the predictions made by the expert and hence can be used as a surrogate for expert predictions. A disadvantage of the expert predictions is that they require a very deep, insight knowledge into the project. The expert making the estimations is very experienced. The time required to create the predictions was negligibly longer than the time required when using the prediction models.

6.2.2. Long-Term Predictions

The values of Mean Magnitude of Relative Errors (MMRE) are presented in Figure 7. The MMRE is calculated using 7 months after the project start (not for the whole project, since it is not yet concluded).

This shows that the best model is using a single, exponential equation. However, by examining only the first 7 months can be misleading,

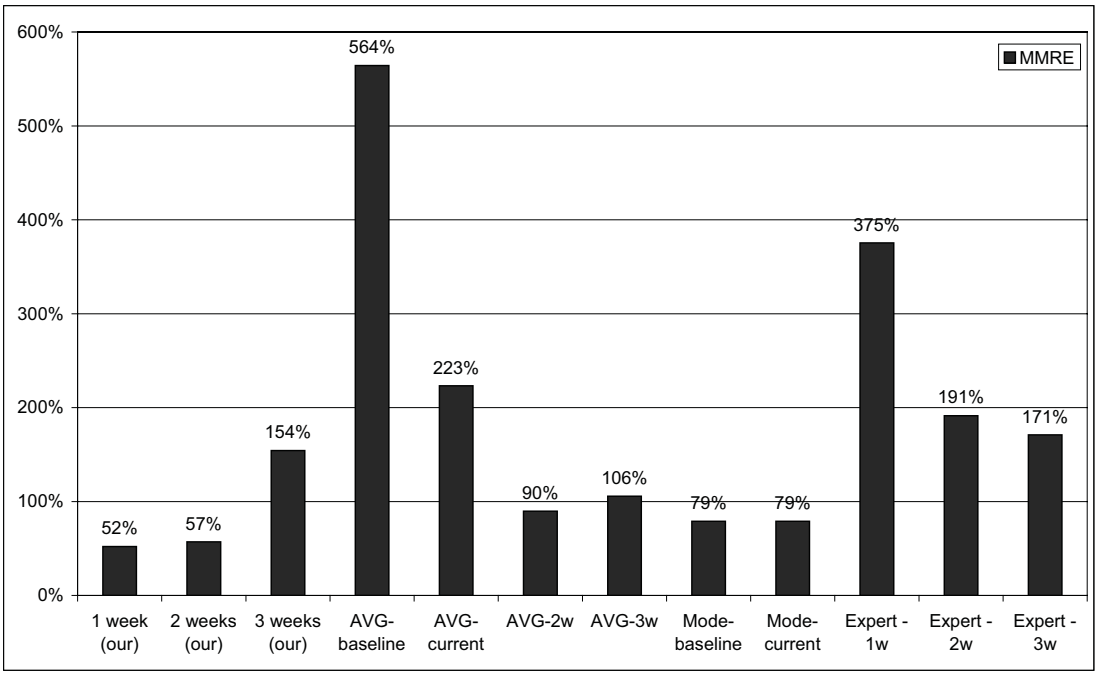


Figure 6. MMRE for short-term predictions

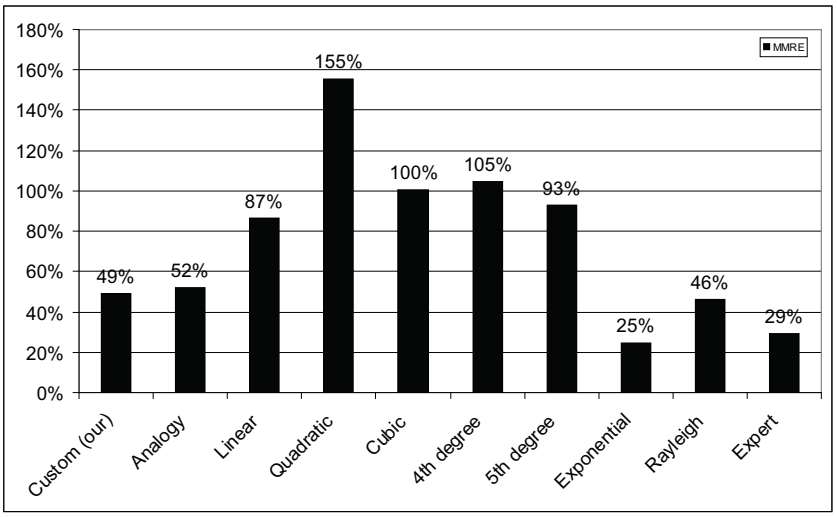


Figure 7. MMRE for the new project

as the predictions for the whole project using exponential equations do not produce trustworthy results by the end of the project. Figure 8 presents the predictions for the whole project.

The results from the stability evaluation are presented in Figure 9. A good model is not changing very much from month to month meaning that the initial predictions are actually trustworthy.

The stability need to be assessed together with the actual defect inflow trend in the project, which is presented in Figure 8. The trend in defect inflow is different that was expected, and it is different than in the baseline project. These differences caused the instability of the model. However, as the project progresses, the peak in month (a+3) was leveraged and the trend in month (b) came back to normal. How-

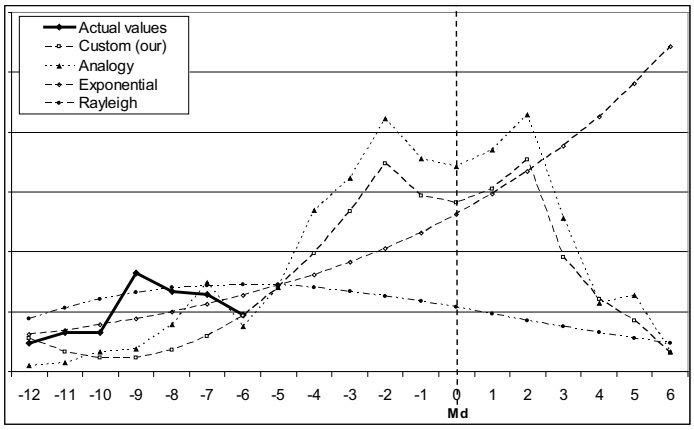


Figure 8. Long-term predictions for the new project

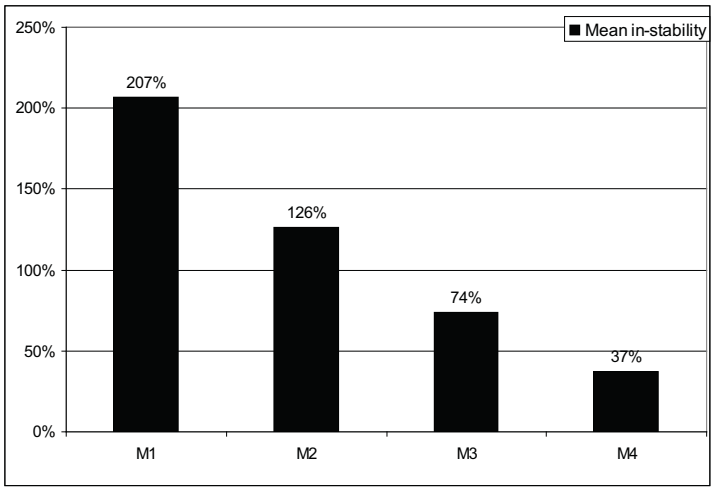


Figure 9. Stability evaluation

ever, given the history and the way in which the scaling factor is calculated, the current predictions are that the defect inflow in the project will be smaller than initially expected.

By observing the MiST chart in Figure 9 we could conclude that the predictions are stable only when there are no peaks in the defect inflow. The peaks are exceptional situations in the projects and they render the predictions unusable, thus call for adjusting the predictions. The presence of such a peak means that the prediction model should be constructed differently – e.g. by choosing a different baseline project, or splitting the baseline project into more curves (c.f. Section 5) – basically the instability is caused by the fact that we use inappropriate curves, which was caused by the fact that this peak was not present in the baseline project.

7. Validity Evaluation

In this section we evaluate the validity of our studies for constructing and evaluation of the methods. We use the framework of Wohlin et al. [24].

The main threat to the *external validity* of our results is the fact that we developed and used prediction methods in a single organization within Ericsson. Even though the organization is a large one (c.f. [20]) and we tested that at more than one product, it could still be seen as a threat. In order to minimize the threat we used the predictions in more than one project and product. The results were sufficiently accurate for both products and they also led to taking immediate actions by project managers in order to prevent potential predicted problems.

The main threat to the *construct validity* is the use of data from manual reporting to construct the prediction models. Although this might be seen as a problem issue, from our discussions with experts it was clear that the simplicity to create the predictions was one of the top priorities and hence our decision. Using test-progress data for predicting defect inflow in the same organization can be found in [19]. In order to minimize the threat that we use "random" variables without empirical causal relationships we preformed a short workshop with Ericsson experts. The outcome of that workshop was that there is empirical causality between the predictors and predicted variables.

Another threat to the construct validity is the use of regression method in our research. Using regression algorithms can be burdened with the problem of overfitting, i.e., fitting the regression equation in short term predictions (or curve for long-term predictions) too closely to the baseline project. Overfitting can cause the models to be inapplicable for other projects. We minimize this threat by evaluating our results in new projects and check the applicability of the models.

The main threat to the *internal validity* of the study is the completeness of the data and mortality of data points. It is rather a common situation in industry that data might be missing due to external factors (e.g. vacations, sick-leaves). In our case the missing data was handled by removing data points which were incomplete and removing the data points which could potentially be affected by low-quality data (mainly during vacation periods). The number of data points removed was small compared to the data sets (less than 5% of data points).

Finally, we have not discovered any threats to the *conclusion validity* as we used established statistical methods to develop the models and confirmed our findings with expert knowledge in the company.

8. Conclusions

This paper presented two complementary methods for predicting defect inflow in large software projects: short-term and long-term defect inflow prediction. The methods are used for the purpose of project planning and monitoring at Ericsson. The goal of introducing new methods in this paper is to provide the experts with support for creating the prediction models using statistical methods based on the data which is already collected in the organization (or which can be collected at reasonable costs). Using linear regression methods resulted in simple and high-cost efficient methods, which could be seen as a trade-off between prediction accuracy and costs of predicting. In this paper we tried to address this trade-off by minimizing the number of measurements to be collected and focus on measurements established and existing in the organization at the same time minimizing the cost for data reconfiguration. However, in the course of our research new ways of data collection were introduced, which improved the practice and allowed for more accurate predictions.

Based on our experience and evaluation of these methods at Ericsson we can recommend using these methods and adjusting them to local needs for particular organizations. The methods are intended to support projects which are structured around work packages and not sub-projects. Our further work is focused on monitoring the performance of these methods in a larger number of projects and further evaluating their robustness. We also intend to deploy these methods to other departments and organizations to check their external validity.

Acknowledgements. We would like to thank Ericsson AB for support in the study, in particular the experts we have the possibility to work with. We would like to thank the Software Architecture Quality Center for support in this study.

References

[1] W. W. Agresti and W. M. Evanco. Projecting software defects from analyzing ada designs. *Software Engineering, IEEE Transactions on*, 18(11):988–997, 1992.

[2] S. Amasaki, T. Yoshitomi, O. Mizuno, Y. Takagi, and T. Kikuno. A new challenge for applying time series metrics data to software quality estimation. *Software Quality Journal*, 13:177–193, 2005.

[3] T. Ball and N. Nagappan. Static analysis tools as early indicators of pre-release defect density. In *27th International Conference on Software Engineering*, pages 580–586, St. Louis, MO, USA, IEEE, 2005.

[4] S. Chulani. Constructive quality for defect density prediction: COQUALMO. In *International Symposium on Software Reliability Engineering*, pages 3–5, 1999.

[5] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings, Menlo Park CA, 1986.

[6] G. H. Dunteman. *Principal Component Analysis*. SAGE Publications, 1989.

[7] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability.

[8] D. Houston, D. Buettner, and M. Hecht. Dynamic COQUALMO: Defect profiling over development cycles. In *ICSP*, pages 161–172, 2009.

[9] M. Klaes, H. Nakao, F. Elberzhager, and J. Munch. Support planning and controlling of early quality assurance by combining expert judgement and defect data – a case study. *Empirical Software Engineering*, 2009.

[10] L. M. Laird and M. C. Brennan. *Software measurement and estimation: a practical approach*. John Wiley and Sons, Hoboken, N.J., 2006.

[11] P. L. Li, J. Herbsleb, and M. Shaw. Forecasting field defect rates using a combined time-based and metrics-based approach: a case study of OpenBSD. In *The 16th IEEE International Symposium on Software Reliability Engineering*, page 10. IEEE, 2005.

[12] Y. K. Malaiya and J. Denton. Module size distribution and defect density. In *11th International Symposium on Software Reliability Engineering*, pages 62–71, San Jose, CA, USA, 2000.

[13] K. Matsumoto, K. Inoue, T. Kikuno, and K. Torii. Experimental evaluation of software reliability growth models. In *The 18th International Symposium on Fault-Tolerant Computing, 1988*, pages 148–153, Tokyo, Japan, IEEE, 1988.

[14] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *26th International Conference on Software Engineering*, pages 282–291. IEEE, 2004.

[15] M. Neil and N. Fenton. Predicting software quality using bayesian belief networks. In *21st Annual Software Engineering Workshop*, pages 217–230, NASA Goddard Space Flight Centre, 1996.

[16] A. M. Neufelder. How to predict software defect density during proposal phase. In *National Aerospace and Electronics Conference*, pages 71–76, Dayton, OH, USA, 2000.

[17] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[18] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin. Capture-recapture in software inspections after 10 years research: Theory, evaluation and application. *Journal of Systems and Software*, 72:249–264.

[19] M. Staron and W. Meding. Predicting weekly defect inflow in large software projects based on project planning and test status. *Information and Software Technology*, 50(7–8):782–796, 2009.

[20] M. Staron, W. Meding, and C. Nilsson. A framework for developing measurement systems and its industrial evaluation. *Information and Software Technology*, 51(4):721–737, 2009.

[21] P. Tomaszewski and L. Lundberg. Software development productivity on a new platform: an industrial case study. *Information and Software Technology*, 47(4):257–269, 2005.

[22] P. Tomaszewski and L. Lundberg. The increase of productivity over time: an industrial case study. *Information and Software Technology*, 48(9):915–927, 2006.

[23] R. E. Walpole. *Probability and statistics for engineers and scientists*. Prentice Hall, Upper Saddle River, NJ, 7th edition, 2002.

[24] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishing, 2000.

e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Automatic Test Cases Generation from Software Specifications

Aysh Alhroob*, Keshav Dahal*, Alamgir Hossain*
*School of Computing, Informatics and Media, University of Bradford
amhalhro@ bradford.ac.uk, k.p.dahal@ bradford.ac.uk, m.a.hossain1@ bradford.ac.uk

Abstract

A new technique is proposed in this paper to extend the Integrated Classification Tree Methodology (ICTM) developed by Chen et al. [13] This software assists testers to construct test cases from functional specifications. A Unified Modelling Language (UML) class diagram and Object Constraint Language (OCL) are used in this paper to represent the software specifications. Each classification and associated class in the software specification is represented by classes and attributes in the class diagram. Software specification relationships are represented by associated and hierarchical relationships in the class diagram. To ensure that relationships are consistent, an automatic methodology is proposed to capture and control the class relationships in a systematic way. This can help to reduce duplication and illegitimate test cases, which improves the testing efficiency and minimises the time and cost of the testing. The methodology introduced in this paper extracts only the legitimate test cases, by removing the duplicate test cases and those incomputable with the software specifications. Large amounts of time would have been needed to execute all of the test cases; therefore, a methodology was proposed which aimed to select a best testing path. This path guarantees the highest coverage of system units and avoids using all generated test cases. This path reduces the time and cost of the testing.

1. Introduction

Unified Modelling Language (UML) provides diagrams to help the software developer to represent different aspects of design. It has become a standard modelling language for designing software. UML represents the system specifications that could be used in software testing. The common definition of software testing usually refers to the testing of program code and not to the testing of models used in earlier development stages of the software development process, such as requirements engineering, analysis or design. Model testing could identify many faults earlier and could, hence, decrease repair costs. A critical component of testing is the construction of test cases. However, software testing is an expensive and labour-intensive process; typically, testing consumes at least 50% of the total costs involved in developing software [7]. Software testing has two important purposes. First, it is commonly used to expose the presence of faults in software. Second, even if testing does not reveal any fault, it still provides increased justification and confidence in the correctness of the software [18]. The Category Partition Method (CPM) was developed by Ostrand and Balcer [19] to generate test cases from functional specifications using the concept of formal test specifications. Several studies [2, 3] have been conducted which focus on CPM. Recently, Chen et al. [14] enhanced the CPM, by means of their choice relation framework. Based on CPM, Grochtmann and Grimm [16] developed a similar but different method – the Classification Tree Method (CTM). Classification trees have been used to construct test cases in the CTM. The absence of a systematic tree construction algorithm is

the major limitation of this method. As a result, users of this method are left with a loosely defined task of constructing a Classification Tree (*Tu*). For complex specifications, this construction task could be difficult, and hence, prone to human error. If a *Tu.* is incorrectly constructed, the quality of the resultant test cases generated from it will be poorly affected. This problem is solved by Chen et al. [13] who use Integrated Classification Tree Methodology (ICTM). This method helps with the identification of test cases via the construction of classification trees. However, their tree constructed method is rather ad hoc. This results in a variation of classification trees constructed in CTM from one software tester to the next, according to his/her personal experience and expertise. A classification hierarchy table (*Hu*) has been used in ICTM to alleviate this problem. The hierarchy table helps to construct classification trees by capturing the hierarchical relation for each pair of classifications.

In ICTM [13], however, a manual method has been used in order to detect the classifications, associated classes and relationships. The manual process requires more human intermediation which can lead to more errors. Manual extraction of information from the software specifications will also increase the cost of testing. In this paper, we propose an approach for generating test cases automatically from software specifications using a class diagram and representing the software constraints by OCL. After decomposing the specifications to functional units, functional units will be represented as a class diagram. An XML schema mapping technique will then be used to read the specification from the class diagram. Transferring data from the XML to build a hierarchy table *Hu.* will reduce human error in building the table. The construction of the *Hu.* is the step before building the classification tree. The paper also proposes a test data refinement technique to discard duplicate sub-trees. The approaches are based on heuristic techniques for determining appropriate test cases for testing software.

Software testing is used to find as many faults as possible so that a piece of software will work to its maximum capabilities. Path testing is a structural testing method that involves using software units to find every possible executable path. Time and cost are the main factors when testing efficiency, therefore, avoidance of lengthy times in testing was the target set after the legitimate test cases were obtained [1]. This paper extends the work presented in Alhroob, Dahal and Hossain [1] to select the best testing path. This technique offers the best path that covers most of the system units.

The rest of the paper is organised as follows. Section 2 presents the previous work in automatic test data generation for UML. Section 3 presents a methodology to generate test data to test class diagram relationships. Section 4 describes the classification tree concepts, whereas the pruning method of duplicate sub-trees is outlined in Section 5. Section 6 covers the best testing path selection. Finally, conclusions and future works are presented in Section 7.

2. Previous Work

Test data are usually generated from the requirements or the code, while the design is rarely concerned with generating test data. Extensible Markup Language (XML) is used to express the constraints on data and detect the rules of software systems. Bertolino et al. [6] used the XML schema to analyse the system specifications and identify the functional units. Categories are identified from functional specifications. The authors [9] used XML schema mapping and category partition to identify the related constraints and relevant values for each category. In general, the test data generation can be extracted from those values and constraints.

Extracting the information from UML diagrams allows the developer to test the system before writing the code. Heuristic techniques can be applied for creating quality test data. Doungsa-ard et al. [15] proposed a GA-based test data generation technique from specifications. Test cases were generated from sequences of triggers for Unified Modelling Lan-

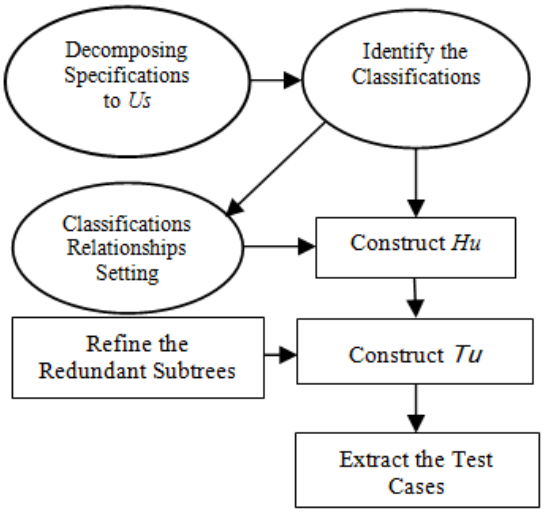


Figure 1. ICTM process

guage (UML) state diagrams. Lia Bao-lin et al. [4] constructed a scenario tree from a sequence diagram. The scenario path was obtained from the tree and the attributes were extracted from the sequence diagram to generate test data automatically. Object Constraint Language (OCL) was used to describe the pre and post conditions for the system to use its system specifications. Sarma et al. [21] proposed a method to extract this information from used case templates, class diagrams and data dictionaries. They also presented an approach to transform the UML sequence diagram to Sequence Diagram Graph (SDG) and provide the SDG with different information necessary to compose test data.

Dehla [23] proposed a technique to generate test data from UML sequence and state diagrams. The main specifications are extracted from the sequence diagram, while the remaining information derives from the state diagram. Sequence diagrams do not provide all information necessary to generate test data automatically. Chen et al. [13] presented ICTM to create test cases from specifications, as shown in Figure 1, via the building of *Hu* and *Tu*. The manual steps, which are indicated by oval shapes in Figure 1, need a software engineering expert. Experts are also needed to decompose the functional unit, identify the classifications and extract the relationships between the classifications. All classifications and their relationships in ICTM are manually entered. The manual processes need more experience, more time and generate higher costs. For large systems there are many classifications and relationships; this requires more effort to input. To avoid the risk and effort, we propose a methodology to enable the manual processes to be done automatically without expert intermediation. Class diagrams will be used to represent the software specifications. The proposed methodology is designed to capture the specifications automatically to build the *Hu* and *Tu*. The building of *Hu* and *Tu* automatically, enables the generation of test cases in an efficient way.

The proposed methodology in this work produced full system coverage test cases, but other issues arose regarding the time needed to execute the test cases, in addition to the cost. Peres et al. [20] introduced a good idea to apply characteristics of software and of testing in test path selection. They used characteristics in path selection strategies, such as complexity, testability and feasibility. They assigned a weight for nodes according to the lower predicate strategy. And the sum of nodes weight was assigned to the branch or path. Emanuela et al. [9] used the degree of similarity between test cases as the main factor for test case selection. This strategy reduced the number of redundant tests and selected the best to execute. Basanieri and co-author [5] proposed a technique to assign

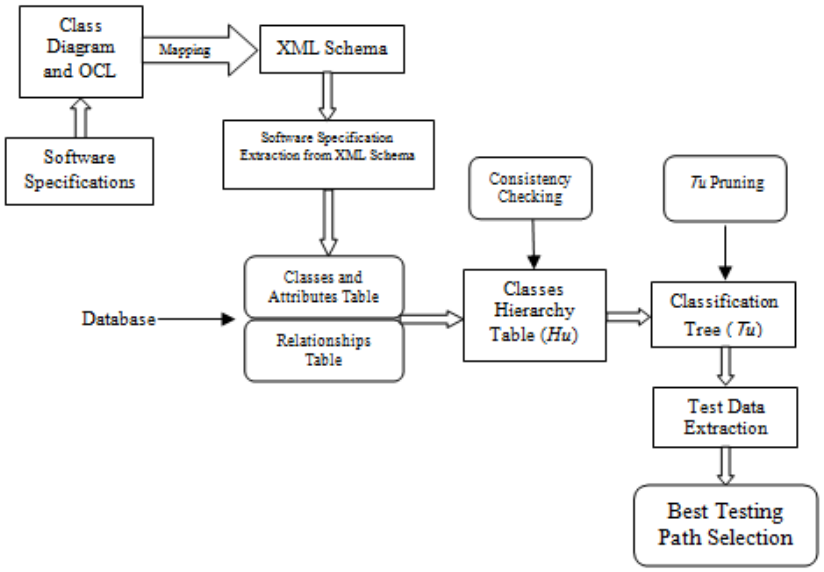


Figure 2. Proposed technique

a weight function to each diagram indicating the functional importance.

Test generation with a verification technology tool [17] extracts the test cases from the UML model. The test cases are selected from a specific objective that a tester would like to test, and can be seen as a specification of a test case. The number of test cases is still large and can be reduced. Our methodology to select the best testing path focused on the maximum coverage percentage and minimum number of test cases. These two factors were treated in previous work [5, 9, 17, 20] separately. In this study, both objectives are considered to achieve the highest results.

3. Methodologies

Class diagram and OCL together represent the functional units. Class diagram mapping to XML code makes the diagram specifications easier to deal with. Due to the variation in design experts, each one will present the functional units using class diagrams in different ways, that is, different class diagrams and different XML codes. To allow the proposed technique to deal with one style of XML, we propose a technique to force different XML to be stored in the same database style.

The specifications will be transferred from the database to construct the *Hu*. The *Hu* will be more reliable due to new consistency checking constraints and automatic information entering. A consistent *Hu* means perfect *Tu*, but control of the number of test cases produced from *Tu* requires more restricted rules to reduce the number of illegitimate test cases. Pruning of the duplication sub-trees will reduce the number of illegitimate test cases. This paper introduces automatic test case generation from software specifications (see Figure 2) and proposes a technique to improve the pruning of sub-trees. There are a large number of legitimate test cases and a technique is needed to select the minimum number to save time and cost. The proposed technique used in this work selects the best path which covers most of the system details.

3.1. UML Class Diagrams

UML class diagrams will be used to represent the component of software specification. The class diagram will represent the functional unit hierarchy relationship. For example, a credit card has two possible types: gold credit card or classic credit card, and each type has its own credit limit, as follows:

– The gold card has two children (credit limit of \$5000] and credit limit of \$6000].

– The classic card also has two children (credit limit \$2000] and credit limit \$3000].

Figure 3 represents the above functional units using three hierarchy classes. OCL is used to represent the constraints and determine the relationship between the attributes in the main class with sub-classes.

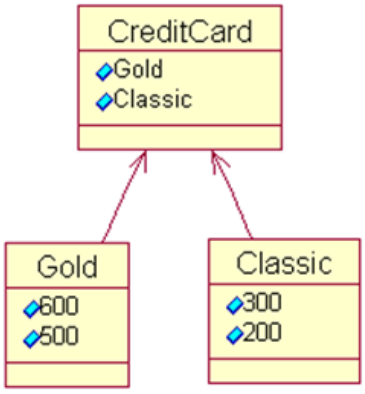


Figure 3. Example of functional unit representation by class diagram

XML mapping is used to extract the software specifications and capture the elements of software from the UML diagram. UML is a standard design modelling language and XML is widely being accepted as an information representation and sharing language across the Internet; efforts have been initiated to map UML diagrams to XML documents [22].

3.2. Automatic Detection for System Specifications

The first three phases of ICTM in Figure 1 are manual phases and the construction of *Hu* is dependent on the relationship setting. A class diagram as a software system model is used to decompose the specification and identify the relationships automatically. Classes, attributes and relationships for the class diagram can be extracted through mapping the diagram to XML. In the proposed approach the XML structure depends on the class diagram design. The diagram design depends on the designer's view, so there is a probability of extracting different XML schema for the same system specifications. Standard XML schema is not our concern, but the specifications in that schema must be captured in a standard way. The extracted classes and attributes will be stored in proper database style, like Table 1. The relationships between classes can be stored in a different table. Now, an automatic transfer technique will be used to transfer

Table 1. Classes and attributes for class diagram in Figure 4

Class	Att1	Att2	Att3
A	$a1$	$a2$	
B	$b1$	$b2$	
C	$c1$	$c2$	
D	$d1$	$d2$	
E	$e1$	$e2$	
F	$f1$	$f2$	
G	$g1$	$g2$	
H	$h1$	$h2$	$h3$
I	$i1$	$i2$	

data from database tables to *Hu*, instead of manually inserting. To make the proposed approach clearer, the following case will be used to represent the main steps of the methodology. Suppose a software tester is given the following of a program *arith-sum*:

1. *arith-sum* has nine input variables A, B, C, D, E, F, H, and I.
2. H has three possible values (denoted by $h1$, $h2$, and $h3$), whereas each of the remaining variables has two possible values (denoted, for example, by $a1$ and $a2$ for A).
3. The input domain of *arith-sum* may contain any combination of possible values from some of these variables, except the following:
   – (A is $a2$) and (B is $b1$ or $b2$)
   – (A is $a2$) and (C is $c1$ or $c2$)
   – (A is $a2$) and (D is $d1$ or $d2$)
   – (A is $a1$) and (E is $e1$ or $e2$)
   – (B is $b2$) and (C is $c1$ or $c2$)
   – (B is $b2$) and (D is $d1$ or $d2$)
   – (B is $b1$ or $b2$) and (E is $e1$ or $e2$)
   – (C is $c2$) and (D is $d1$ or $d2$)
   – (C is $c1$ or $c2$) and (E is $e1$ or $e2$)
   – (C is $c1$ or $c2$) and (F is $f2$)
   – (C is $c1$ or $c2$) and (H is $h1$, $h2$, or $h3$)
   – (D is $d1$ or $d2$) and (E is $e1$ or $e2$)
   – (D is $d1$ or $d2$) and (F is $f2$)

- (D is $d1$ or $d2$) and (H is $h1$, $h2$, or $h3$)
- (E is $e1$ or $e2$) and (G is $g1$ or $g2$)
- (F is $f2$) and (G is $g1$ or $g2$)
- (F is $f1$) and (H is $h1$, $h2$, or $h3$)
- (G is $g1$ or $g2$) and (H is $h1, h2$, or $h3$)

4. *arith-sum* calculates the arithmetic sum of those variables entered.

Suppose we simply define the classes as the input variables and the attributes as the possible values. For example, A is taken as a class with $a1$ and $a2$ as its attributes. Then Figure 4 shows the class diagram for *arith-sum*.

3.3. Classes Hierarchy Table

Automatic construction of a classification hierarchy table, *Hu* with class relationships for each pair of classes is the main target in this section. There are four possible types of hierarchical relationships, as follows [10, 13]:

1. Class $[X]$ is a loose ancestor of class $[Y]$ (denoted by $[X] \Leftrightarrow [Y]$.
2. Class $[X]$ is a strict ancestor of $[Y]$ (denoted by $[X] \Rightarrow [Y]$). A black arrow means direct relation, but red indicates an indirect relation.
3. Class $[X]$ is incompatible with Class $[Y]$ (denoted by $[X] \sim [Y]$).
4. Class $[X]$ has other relations with Class $[Y]$ (denoted by $[X] \bigotimes [Y]$).

The conditions associated with each of the above hierarchical relations are commonly exclusive and exhaustive. These hierarchical relations are used to determine the relative position of $[X]$ and $[Y]$ in *Tu*. For example, $[X] \Rightarrow [Y]$ corresponds to the situation where $[X]$ will appear as either a parent or an ancestor of $[Y]$ in *Tu*; in current work the loose ancestor relationship was discarded. Figure 5 depicts the completed *Hu* Every element in it contains a hierarchical operator and corresponds to the hierarchical relations between a pair of classifications.

3.4. Consistency Checking

Cain et al. [8] introduced the consistency problem and proposed a technique to detect inconsistency relations. Let t_{ij} denote the element at the i^{th} row and the j^{th} column of Figure 5. Consider t_{12} and t_{21} in Figure 5. They correspond the $[A] \Rightarrow [B]$ and $[B] \bigotimes [A]$, respectively. Suppose,

- t_{21} constraints are entered before that for t_{12},
- t_{21} constraints are entered correctly, causing "$\bigotimes$" to t_{21} to assign the hierarchical operator,
- a mistake has been made during the entry of the constraints for t_{12}, causing incorrect assignment of the hierarchical operator "$\sim$" to t_{12}.

As we note that the error is unwanted, to recover this problem, a methodology is proposed in this paper to ensure that all of the relationships are entered in a systematic way and no conflict occurs between them. We need the following five conditions to do that:

1. For e_{ij} (where e_{ij} is corresponding for all classes) we have to detect the relationships automatically for each pair of classes (A, B) in *Hu*.
2. $X_i \rightarrow Y_i$, where X and Y are two associated classes in the database.
3. If $A = X$ and $B = Y$, then A is a child of B, and all of the attributes of A are related with at least one of the B attributes.
4. $A = X$ and $B \neq Y$, then A is incompatible with Y.
5. If $A = Y$ and $B = X$, then A is a parent of B, and at least one of the A attributes are related with all B's attributes.

4. Classification Tree (*Tu*)

Based on a predefined tree construction algorithm [13], the corresponding *Tu* can be automatically constructed from the *Hu* in Figure 5. The tree represents the relationships between the classes and determines the parents and children of classes. The classification tree is used to generate the test cases; if the test cases cover 100% of the tree branches that means all parents, children and attributes will be tested. Complete or legitimate test case extraction is our target. Human error in the specification ex-

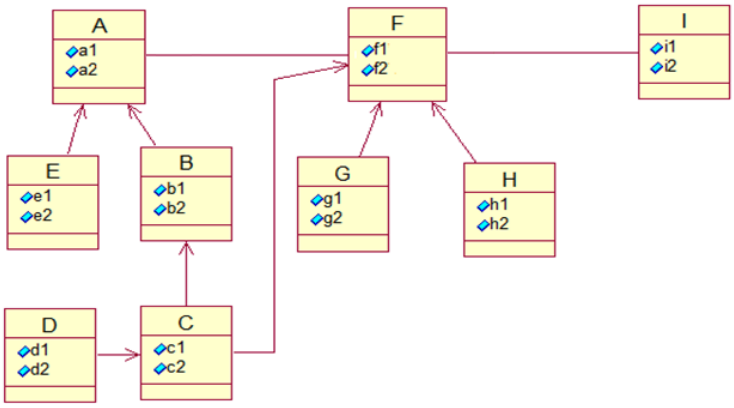


Figure 4. Class diagram for *arith-sum*

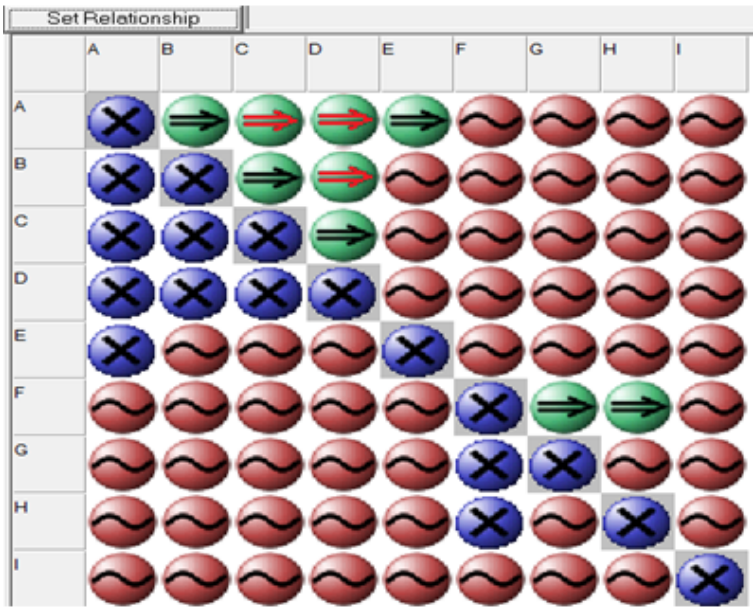


Figure 5. Classification-Hierarchy table (*Hu*)

traction phase is avoided by using the proposed automatic methodology.

Occasionally, a classification tree may not be able to reflect all the constraints between classifications. Therefore, all potential test cases constructed from the classification tree should be verified with the specification. In this way, we can classify and remove the potential test cases that deny the specification. Such potential test cases are known as illegitimate test cases. Chen and Poon [11] proposed that the final purpose of the classification tree method is to construct legitimate test cases, and the classification tree is just a means for this construction. Given a classification tree *Tu*, let N^i and N^t be the number of potential test cases and legitimate test cases, respectively. Chen et al. [12] defined an effectiveness metric, E_p for *Tu* as the follows:

$$E_P = \frac{N^t}{N^i} \tag{1}$$

For more illustration, for equation (1), let $N^i = 40$ and $N^t = 5$, then $E_p = 0.125$. N^t can only be known after removing all illegitimate test cases from the set of possible test cases. Obviously, a small value of E_p is undesirable, as effort will be wasted on illegitimate test cases. The existence of duplicate sub-trees under different top-level classifications in a classification tree is a main cause of a poor E_p.

From this remark, Chen and Poon developed a tree restructuring algorithm to remove duplicates, to obtain a better value of E_p for classification trees with duplicate sub-trees under different top-level classifications [12]. Deleting the duplicate sub-tree will cause the illegitimate test cases to be discarded and will increase the effectiveness metric of the classification tree. Determining the duplication availability and choosing a suitable sub-tree to delete are main factors for duplication sub-tree pruning.

5. Pruning the Duplication Subtrees

Chen et al. [12] proposed an algorithm to avoid duplicate sub-trees and improve the value of E_P. They observed that the algorithm for removing duplicates has many limitations, such as dealing with duplicate sub-trees under different top-level classifications and the fact that only one set of duplicate sub-trees can be removed from the classification tree at any one time. To overcome these limitations, we propose a restructuring algorithm for pruning the duplicate sub-tree, hence, improving the value of E_p. The proposed algorithm will deal with duplicate sub-trees and choose the best ones to keep with Tu and remove the others. The algorithm will compare every sub-tree (ST) in Tu with others to detect duplications in same and different top-level classifications (P_i). The duplicate sub-tree suitable for deleting is one that produces a large number of test cases by integrating with others under the same or different top level. The following methodology illustrates the detection and deletion of suitable duplicate sub-trees.

1. P_1, P_2 and P_3 are top level class, for example, they correspond to A, F and I respectively as shown in Figure 6, where $P_i \geq 1$.
2. The Tu has been divided into levels L_1, L_2,..., L_i, where $L_i \geq 1$. Every L_i has a value depending on the level number, for example L_1 has a value 1, L_2 has a value 2, etc.
3. Every classification (X) in Tu has its own principle value sequentially, for example ($A = 1$, $B = 2$,..., $X_n = V$). Each children (x) for X will take the same value of X. $x_v = L_i * X_n$, where x_v is the value of each child.
4. We propose (2), (3) and (4) equations to calculate some of parameters values to determine which trees in the system must be deleted.

$$X_V = \sum_{n=1}^{n} x_v \tag{2}$$

$$Q_P = \sum_{n=1}^{n} X_v \tag{3}$$

$$Q_{ST} = \sum_{n=1}^{n} X_{Dv} \tag{4}$$

$$RQ_{ST}(\textit{Ratio of duplication } ST \textit{ in } P_i) = \frac{Q_{ST}}{Q_P} \tag{5}$$

Where X_v, Q_P, Q_{ST} and X_{DV} are the values of X, P_i, duplicate ST and duplicate classes, respectively. We observe that if the ratio of Q_{ST} on Q_P in equation (5) is smaller, then the number of branches in P_i be big, which means more test cases will be generated from this P_i, If we delete the duplication ST from P_i which has more branches, we avoid generating more illegitimate test cases. For example, if RQ_{ST} in P_1 < RQ_{ST} in P_2 the duplication ST in P_1 must be deleted. The algorithm repeats the above process until there are no duplicated STs across any pairs of distinct top-level sub-trees. This algorithm deals with all duplicated STs in Tu whether in the same top level or different and treats duplications for two sub-trees or more. By referring to Figure 4, the classification tree in Figure 6 contains duplications for C ST. The C sub-tree arises in two places, the first one occurs under the B class and the second one under the F class; the algorithm will detect the duplica-

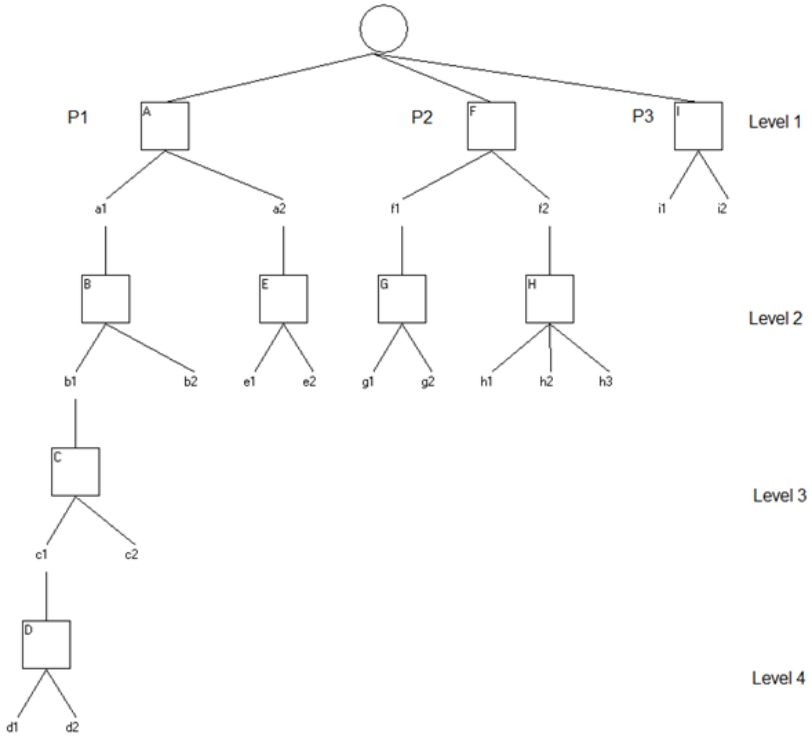


Figure 6. *arith-sum* Classification Tree

tion by comparing classifications with others in Tu even in different top level classes. If the duplication is detected, the algorithm starts to capture the suitable ST to delete. For example, as noted in the class diagram the C class is associated with B and F; that means that the C sub-tree should appear under two parents of classes. Duplication will occur even under B or F. One of the sub-trees must be chosen for deletion depending on the proposed sub-tree pruning algorithm.

In this case, and by referring to equation (4), the duplicate sub-tree that comes under F is selected for deletion. From Tu of *arith-sum* in Figure 6, a total of 60 potential test cases can be constructed; some of the test cases are shown in Figure 7. The total number of test cases, before deleting the duplicate sub-trees, was 108, so the classification tree pruning technique deleted 48 illegitimate test cases by checking the 60 legitimate potential test cases against the specification of *arith-sum*. 32 potential test cases were found to be illegitimate and therefore removed. For example, the potential test cases 5–10 were illegitimate because class $(F) = f2$ cannot coexist with class $(C) = c1$ and $c2$.

6. Best Testing Path Selection

Testing is the process of executing a system with the intention of finding errors. Assume that there are 5 possible paths with loop<10, its equal 10^7 different execution flows. If we executed one test per millisecond, it would take 1.585 years to test this system. The proposed methodology in this work aims to select a best testing path. This path guarantees the highest coverage of system units. Each test case generated for the class diagram in Figure 3 represents a test path. Before continuing to explain the technique, we have to differentiate between the test path and the case. The test case is the combination of the node attributes, in contrast, the test path is the nodes that are shared in a test case, i.e. Test case 2 in Figure 7 is $A = a1$, $B = b1$, $C = c1$, $D = d1$, $F = f1$, $G = g1$, $I = i2$ and the test path 2 is A, B, C, D, F, G, I.

Set Relationship | Classification Tree | Potential Test Frames

Frame 1	A = a1	B = b1	C = c1	D = d1	F = f1	G = g1	I = i1
Frame 2	A = a1	B = b1	C = c1	D = d1	F = f1	G = g1	I = i2
Frame 3	A = a1	B = b1	C = c1	D = d1	F = f1	G = g2	I = i1
Frame 4	A = a1	B = b1	C = c1	D = d1	F = f1	G = g2	I = i2
Frame 5	A = a1	B = b1	C = c1	D = d1	F = f1	H = h1	I = i1
Frame 6	A = a1	B = b1	C = c1	D = d1	F = f1	H = h1	I = i2
Frame 7	A = a1	B = b1	C = c1	D = d1	F = f1	H = h2	I = i1
Frame 8	A = a1	B = b1	C = c1	D = d1	F = f1	H = h2	I = i2
Frame 9	A = a1	B = b1	C = c1	D = d1	F = f1	H = h3	I = i1
Frame 10	A = a1	B = b1	C = c1	D = d1	F = f1	H = h3	I = i2

Figure 7. Some of potential test cases generated from *Tu arith-sum* (10 out of 60)

The best testing path technique, which covers maximum units, concerns two main aspects.

Firstly, the weight of each class (node) is dependent on the number of attributes and its level. The level of node will be affected by the number of participant nodes and branches, i.e. H node in Figure 6 has a weight of 12. The weight of H has been calculated by equation (2) and $x_v = L_i * X_n$, where x_v is the value of each child. Secondly, the weight of each path is the weight of all nodes that share one path, i.e. test case 1 in Figure 7 goes through A, B, C, D, F, G and I nodes. The weight of each node is as follows: $A = 1, B = 4, C = 6, D = 8, F = 4, G = 8$ and $I = 6$. The weight of the path is calculated as follows:

$$Pw = \sum_{n=1}^{n} X_v \tag{6}$$

where is the Pw is the weight of path and X_v is the weight of each node. One of the case studies used in this work is the ATM machine. This system contains 18 nodes and each node has its own attributes. Table 2 represents each node weight that has been calculated automatically. As noted, the nodes are called by their numbers and not by names, because the testing path selection methodology deals with nodes by their numbers, i.e. CardReader = 1, Inquiry = 2, Deposit = 3, ... etc.

Figure 8 represents the ATM system nodes; the Node Tree (NT) shows the hierarchy relationships between the nodes. The tree represents the relationships between the classes and determines the test paths. If the test paths cover 100% of the tree nodes that means all system units will be tested.

Table 2. ATM Nodes Weight

N_i	Weight	N_i	Weight
1	2	10	60
2	12	11	66
3	18	12	96
4	32	13	130
5	20	14	112
6	48	15	225
7	14	16	160
8	32	17	68
9	54	18	36

In this phase, the proposed methodology extracted the test paths automatically based on the proposed test path construction algorithm. 15 test paths were generated from NT (see Table 3), one is the best one. It is not necessary that the testing path covers all units of the system; any test path that covers the highest percentage of systems is the best one. The highest percentage does not mean the largest number of nodes. Each node has its own weight, as illustrated in Table 2. The best path, that has the highest weight out of the total weight, is P_9, as shown in Table 3. P_9 covers 10 nodes out of 18 with 51% system details coverage.

In fact, one test path cannot cover all units, as there may be many paths/loops. To improve the coverage of the system, we propose selecting the second best testing path as well to support the first best testing path. The selection method for the second best testing path depends on the non-similarity of nodes contained in the

Table 3. Testing Paths Weight(TPW)

P_i	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	TPW
P_1	1	2	3	4	7	8	9	18	–	–	200
P_2	1	2	3	5	7	8	9	18	–	–	188
P_3	1	2	3	6	7	8	9	18	–	–	216
P_4	1	2	3	4	7	10	11	12	13	18	466
P_5	1	2	3	5	7	10	11	12	13	18	454
P_6	1	2	3	6	7	10	11	12	13	18	482
P_7	1	2	3	4	7	10	11	14	15	18	577
P_8	1	2	3	5	7	10	11	14	15	18	565
P_9	1	2	3	6	7	10	11	14	15	18	593
P_{10}	1	2	3	4	7	10	11	14	16	18	512
P_{11}	1	2	3	5	7	10	11	14	16	18	500
P_{12}	1	2	3	6	7	10	11	14	16	18	528
P_{13}	1	2	3	4	7	10	17	18	–	–	242
P_{14}	1	2	3	5	7	10	17	18	–	–	230
P_{15}	1	2	3	6	7	10	17	18	–	–	258

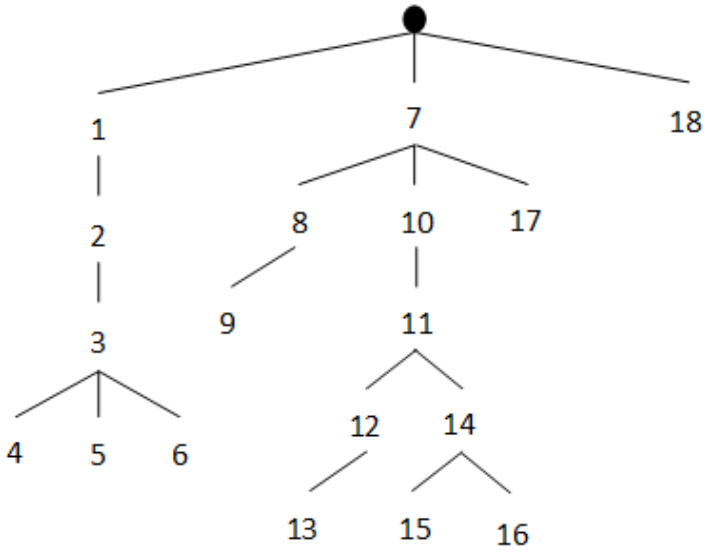


Figure 8. ATM Nodes Tree

best testing path. In other words, we want to select a second best testing path, which contains as many different nodes as possible compared to the first best testing path. Emanuela et al. [12] used the similarity function to reduce the test cases. To select the second best path, we used node non-similarity criterion. Based on the non-similarity degree between the best path and others, the testing paths eliminated were those with the biggest similarity degree. There is a probability for getting more than one test path with the same degree of non-similarity criterion; the highest weight of non-similar nodes is the factor used to choose one of them.

In ATM testing path there are four paths (P_1, P_2, P_4 and P_5) met the highest non-similar criterion, non-similar path with the highest

Table 4. Non-similar paths weight

P_i	Non-similar Nodes	Weight
P_1	4, 8 and 9	118
P_2	5, 8 and 9	106
P_3	6, 8 and 9	134
P_4	4, 12 and 13	285

nodes weight is selected. Table 4 represents those paths with non-similar node weights. P_3 has two conditions necessary (non-similarity and highest weight) for being chosen as the second best testing path. Both testing paths (best and second best testing path) cover more than 74% of system details.

7. Conclusion

In this paper the ICTM has been improved to detect specifications automatically. These specifications are used to generate test cases. To control the consistency of relationships in *Hu*, we proposed an algorithm to enter the class hierarchy relationships in a systematic way. Consistency relationship entering techniques support the reliability of ICTM.

In this paper, a restructured algorithm was proposed to remove duplication sub-trees, either at the same top level or at different top levels. This technique can offer more pruning and produce legitimate and non-duplicated test cases. Testing path selection was one of the concerns in this work in order to reduce the number of expectation flows. Test paths have been determined and one has been selected automatically to be the best among them. The best testing path covers most of the system units and avoids the undesirable time needed to execute all test paths. To improve the percentages of coverage, we propose the selection of additional test paths based on dissimilarity to the best test path.

In future work we will conceder to use class diagrams, OCL and sequence diagrams to represent software specifications to provide other additional information. Therefore, by combining these two UML specifications in future work we will be able to capture most of the system specifications.

References

[1] A. Alhroob, K. Dahal, and A. Hossain. Automatic test cases generation from software specifications modules. In *Proceedings of the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques*, pages 130–142. Springer, 2009.

[2] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Systems Integrity, Software Safety and Process Security: Building the System Right*, pages 3–10, Gaithersburg, MD, USA, IEEE Press, 1992.

[3] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 69–79, Gaithersburg, MD, USA, IEEE, 1994.

[4] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong. Test case automate generation from UML sequence diagram and OCL expression. In *Proceedings of the 2007 International Conference on Computational Intelligence and Security: CIS*, pages 1048–1052, 2007.

[5] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite approach to planning and deriving test suites in UML projects. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002–the Unified Modeling Language*, pages 383–397. Springer, 2002.

[6] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Automatic test data generation for XML schema-based partition testing. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 4. IEEE Computer Society, 2007.

[7] B. Boris. *Software testing techniques*. Van Nostrand Reinhold Co, second edition, 1990.

[8] A. Cain, T. Y. Chen, D. Grant, P. L. Poon, S. F. Tang, and T. H. Tse. An automatic test data generation system based on the integrated classification-tree methodology. *Software Engineering Research and Applications*, pages 225–238, 2004.

[9] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Automated test case selection based on a similarity function. In *Workshop Modellbasiertes Testen (MOTES07)*, Bremen, 2007.

[10] T. Y. Chen and P. L. Poon. Classification-hierarchy table: a methodology for constructing the classification tree. In *Proceedings of the 1996 Australian Software Engineering Conference*, page 93, Washington, DC, USA, IEEE Computer Society, 1996.

[11] T. Y. Chen and P. L. Poon. Improving the quality of classification trees via restructuring. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*, page 83, 1996.

[12] T. Y. Chen, P. L. Poon, and T. H. Tse. A new restructuring algorithm for the classification-tree method. In *Proceedings of the Software Technology and Engineering Practice*, pages 105–114, 1999.

[13] T. Y. Chen, P. L. Poon, and T. H. Tse. An integrated classification-tree methodology for test case generation. *International Journal of Software Engineering and Knowledge Engineering*, 10(6):647–679, 2000.

[14] T. Y. Chen, P. L. Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. *IEEE transactions on software engineering*, 29(7):577–593, 2003.

[15] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart. *Advanced Design and Manufacture to Gain a Competitive Edge*, chapter GA-based for Automatic Test Data Generation for UML State Diagrams with Parallel Paths, pages 147–156. Springer, London, 2008.

[16] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[17] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[18] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE transactions on Software Engineering*, 18(1):33–43, 1992.

[19] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[20] L. M. Peres, S. R. Vergilio, M. Jino, and J. C. Maldonado. Path selection in the structural testing: Proposition, implementation and application of strategies. In *Proceedings. XXI Internatinal Conference of the Chilean Computer Science Society*, pages 240–246. SCCC, 2001.

[21] M. Sarma, D. Kundu, and R. Mall. Automatic test case generation from UML sequence diagram. In *Proceedings of the 15th International Conference on Advanced Computing and Communications*, pages 60–67, Washington, DC, USA, 2007. IEEE Computer Society.

[22] J. Singh. Mapping UML diagrams to XML. Master's thesis, Jawaharlal Nehru University New Delhi, India, 2003.

[23] D. Sokenou. Generating test sequences from UML sequence diagrams and state diagrams. *Informatik für Menschen*, 2(94):236–240, 2006.

e-Informatica Software Engineering Journal (http://www.e-informatyka.pl/wiki/e-Informatica) is an international journal that concerns theoretical and practical issues pertaining development of software systems, and focuses on experimentation in software engineering.

The purpose of e-Informatica is to publish original and significant results in all areas of software engineering research.

The scope of e-Informatica includes methodologies, practices, architectures, technologies and tools used in processes along the software development lifecycle, but particular stress is laid on empirical evaluation.

Topics of interest include, but are not restricted to:

- Software requirements engineering and modeling
- Software architectures and design
- Software components and reuse
- Software testing, analysis and verification
- Agile software development methodologies and practices
- Model driven development
- Software quality
- Software measurement and metrics
- Reverse engineering and software maintenance
- Empirical and experimental studies in software engineering
- Evidence based software engineering
- Systematic reviews
- Object-oriented software development
- Aspect-oriented software development
- Software tools, containers, frameworks and development environments
- Formal methods in Software Engineering.
- Internet software systems development
- Dependability of software systems
- Human-computer interface
- AI and knowledge based software engineering
- Project management

The submissions will be accepted for publication on the base of positive reviews done by international Editorial Board (http://www.e-informatyka.pl/wiki/e-Informatica_-_Editorial_Board) and external reviewers. English is the only accepted publication language. To submit an article please enter our online paper submission site.

Subsequent issues of the journal will appear continuously according to the reviewed and accepted submissions.

http://www.e-informatyka.pl/wiki/e-Informatica

e-Informatica

ISSN 1897-7979