e-Informatica

software engineering journal

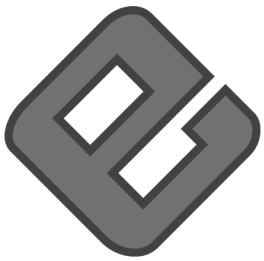
2008 volume 2 issue 1

e-Informatica

e-Informatica

software engineering journal

2008 | volume 2 | 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.

Editorial Board

Contents

Editorial

Zbigniew Huzar, Lech Madeyski 7

Papers

Trustworthy Assembly of Components Using the B Refinement

Arnaud Lanoix, Jeanine Souquières 9

Computation Independent Representation of the Problem Domain in MDA

Janis Osis, Erika Asnina, Andrejs Grave 25

Integrating Human Judgment and Data Analysis to Identify Factors Influencing Software Development Productivity

Adam Trendowicz, Michael Ochs, Axel Wickenkamp, Jürgen Münch, Yasushi Ishigai, Takashi Kawaguchi 41

A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams

Philip Samuel, Rajib Mall 61

Editorial

It is a pleasure to present to our readers the second 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 e-Informatica Software Engineering Journal includes methodologies, practices, architectures, technologies and tools used in processes along the software development lifecycle, but particular interest is in empirical evaluation.

The second issue of the e-Informatica Software Engineering Journal includes four papers carefully reviewed by Editorial Board members, as well as by external reviewers, and then selected by the editors. The first of the papers by Lanoix and Souquières suggest to exploit existing notations, languages and tools to specify the behavior of components and propose to use of B assembling and refinement mechanisms to easy the verification of the interoperability between interfaces and the correctness of the component assembly. The second paper by Osis et al. proposes Topological Functioning Modeling for Model Driven Architecture approach which increases the degree of formalization, introduces more formal analysis of the problem domain, enables defining what the client needs, verifying textual functional requirements, and checking missing requirements in conformity with the domain model. The third paper by Trendowicz et al. proposes a novel approach for identifying the most relevant factors influencing software development productivity. The last paper by Samuel and Mall presents a novel technique for test case generation using dynamic slicing of UML sequence diagrams.

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 2, Issue 1, 2008

Trustworthy Assembly of Components Using the B Refinement

Arnaud Lanoix*, Jeanine Souquières*
*LORIA – Nancy-Université, CNRS
Arnaud.Lanoix@loria.fr, Jeanine.Souquieres@loria.fr

Abstract

In component-based software development approaches, components are considered as black boxes, communicating through required and provided interfaces which describe their visible behaviors. In the best cases, the provided interfaces are checked to be compatible with the corresponding required interfaces, but in general, adapters have to be introduced to connect them. We propose to exploit existing notations and languages with their associated tools to specify working systems out of components: UML composite structure diagrams to express the architecture in terms of components and their interfaces, class diagrams, sequence diagrams and protocol state machines to describe the behavior of each component. Component interfaces will then be expressed in B in order to verify the interoperability. The use of B assembling and refinement mechanisms eases the verification of the interoperability between interfaces and the correctness of the component assembly.

1. Introduction

Recent works have shown that assembling components independently produced and taking into account the verification of their assembly with appropriate tools is a promising approach developped since the nineties [14, 21]. The underlying idea is to develop software systems by assembling existing parts [8, 33], as it is common in other engineering disciplines, such as electrical or mechanical engineering. Among the advantages of such approaches, we can cite: (i) reusability of trustworthy software components, (ii) reduction of the development costs due to the reusability, and (iii) flexibility of systems developed by this approach. Assembling components needs to be supported by design methods and verification tools: on one hand, current technologies of components [25, 28, 32, 34] do not take into account safety requirements, on the other hand, development and certification processes of critical software, based on formal methods, is not well suited to component-based approaches.

The development of component-based systems introduces a fundamental evolution in the way systems are acquired, integrated, deployed and modified. Systems can be designed by examining existing components, like COTS or Commercial Off-The-Shelf components, to see how they meet the expected requirements and decide how they can be integrated to provide the expected functionalities. Next, the system is engineered by assembling the selected components with some locally developed pieces of code [10, 20].

Components are seen as black-boxes units which only specify interfaces and explicit dependencies. An interface describes services offered and required by a component without disclosing the component implementation. Component interfaces are the only access to component informations and functionalities. The services offered by a component are described by provided interfaces and the needed services are described by required interfaces.

For different components to be deployed and to work together, they must interoperate: their

interfaces must be compatible through different levels of compatibility depending on the requirements of the developed system. The syntactic level covers signature aspects of attributes and methods provided or required by the interfaces whereas the semantic level concerns behavioral aspects of the considered methods and the protocol level covers the allowed sequence of method calls.

The availability of formal languages and tool support for specifying interfaces and checking their compatibility is necessary in order to verify the interoperability of components. Our approach is supported by a rigourous development methodology based on UML and the B method and is introduced at the level of software architecture. The idea to define component interfaces using B has been introduced in an earlier paper [12]: semantics and protocols of the component services can be easily modeled using the B formal method. The use of the B refinement [1] to prove that two components are compatible at the signature and semantics levels has been explored in [11]. To guarantee a trustworthy assembly of components, each connection of a required interface with another provided interface has to be considered. In the best cases, the provided interface constitutes an implementation of the required interface. In general cases, to construct a working system out of components, adapters have to be defined. An adapter is a piece of glue code that expresses the mapping between a required and a provided interface. At the signature level, it must express the mapping between required and provided variables and how the required methods are implemented in terms of the provided ones. In [26], we have proposed a first definition of an adapter in a simple case, with only one required and one provided interface.

In this paper, we generalize the previous results, taking into account a more general assembly of components with the use of both cases of interfaces for different components to be connected. We use the following notations:

- UML 2.0 [27] composite structure diagrams serve to express the overall architecture of the system in terms of components and interfaces.
- UML 2.0 class diagrams serve to express interface data model with its different attributes and operations.
- The usage protocol of each interface can be modeled by a Protocol State Machine (PSM).
- UML 2.0 sequence diagrams serve to express the interactions between the components to be connected.
- The use of the formal method B [1] and its associated tools serve to specify interfaces, giving a special attention to correctness, increasing confidence in the developed systems: correctness of specifications, as well as correctness of the followed process with verification aspects.

In the following, we present the case study of a simple access control system defined in terms of components with a special focus on the identification component, itself defined in terms of components. Section 3 exposes the trustworthy assembly problem in a general manner. Section 3 presents a simple case of trustworthy component assembly. Section 4 presents a more general case of component assembly. Some related works are discussed in Section 5 and Section 6 concludes the paper.

2. Case Study: a Simple Access Control System

We illustrate our purpose with the case study of a simple access control system which manages the access of authorized persons to existing buildings [2]. Persons who are authorized to enter the building have to be identified. The needed authentication informations may be stored on an electronic access card or a sophisticated key or a bar code pass, etc. Turnstiles block the entrance and the exit of each building until an authorization is given whereas identification systems are installed at each entrance and exit of the concerned buildings. The means of identification can be read to kept out the authentication informations. It can be inserted and ejected and must be taken by the user before a fixed time of 30 seconds, else it is retracted and kept by the system.

A partial view of the architecture of the access control system is given Figure 1 as a UML 2.0 composite structure diagram. Such diagrams

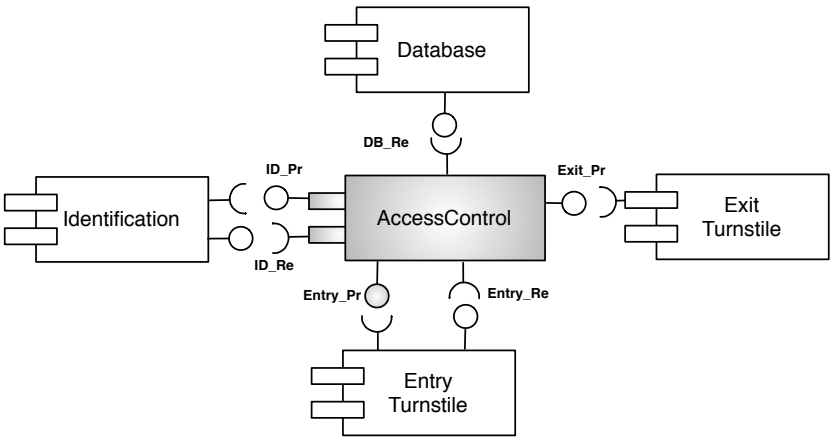


Figure 1. Partial view of the architecture of the access control system

contain named rectangles corresponding to the components of the system; here, we have depicted four components: the AccessControl component corresponding to the system requirements, an Identification component corresponding to the control of the identification, a Database component which is a passive component knowing informations about the authorization of each concerned user and some Turnstile components. They are connected by means of interfaces which may be required or provided. Required interfaces explicit the context dependencies of a component and are denoted using the "socket" notation whereas provided interfaces explain which functionalities the considered component provides and are denoted using the "lollipop" notation.

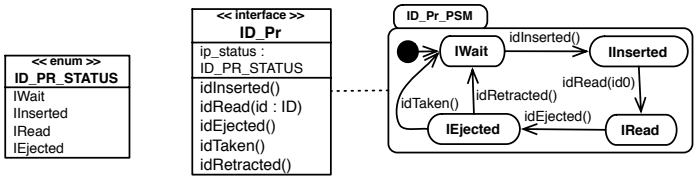
We will focus on the interactions between the AccessControl and the Identification components. Requirements concerning the Identification component are expressed by the two interfaces of the AccessControl component. They have been outlined in Figure 2; they are modeled by class diagrams with their different attributes and methods:

- ID_Pr corresponds to its provided interface related to the Identification component with five operations: idInserted, idRead parameterized by some authentication informations represented by id, id Ejected, idTaken and idRetracted.
- ID_Re corresponds to its required interface which must be provided by an Identification component with three operations, readId, acceptedId and refusedId.

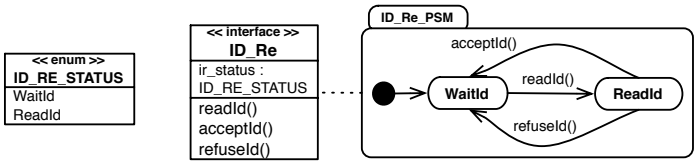
Enumerated data types are defined using the stereotype "enum". The usage protocol of each interface is modeled by a Protocol State Machine (PSM) as presented in Figure 2. A PSM specifies the external behavior of the component, with the order of the allowed method calls starting from its initial state.

The B method. It is a formal software development method based on set theory which supports an incremental development process, using refinement [1]. A development begins with the definition of an abstract model, which can be refined step by step until an implementation is reached. The refinement of models is a key feature for incrementally developing models from textual descriptions, preserving correctness in each step.

The method has been successfully applied in the development of several complex real-life applications, such as the METEOR project [4]. It is one of the few formal methods which has robust and commercially available support tools for the entire development life-cycle from specification down to code generation [5]. The B method provides structuring primitives that allow one to compose machines in various ways. Large systems can be specified in a modular way and in an object-based manner [22, 24]. Proofs of invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [31] or B4free [13], an academic version of AtelierB. Checking proof obligations with B support tools



(a) Provided interface ID_Pr



(b) Required interface ID_Re

Figure 2. Interfaces of AccessControl related to Identification

is an efficient and practical way to detect errors introduced during development.

Example. For each interface given in Figure 2, we give a B model as presented in Figure 3: each model consists of a set of variables, invariant properties of those variables and operations. The state of the model, i.e. the set of variables values, is modifiable by operations, which must preserve its invariant:

- in the ID_Pr model, the variable id_status has four possible states and its initial state is IWait. After an idInserted() call, its state is changed to IInserted,
- the ID_Re model expresses the required behaviors of the card reader. A variable, namely ip_status, gives its state, which is initialized with WaitId.

In an integrated development process, the B models can be obtained by applying systematic derivation rules from UML to B [23, 24].

Our purpose is to define the AccessControl component using existing components available on the market.

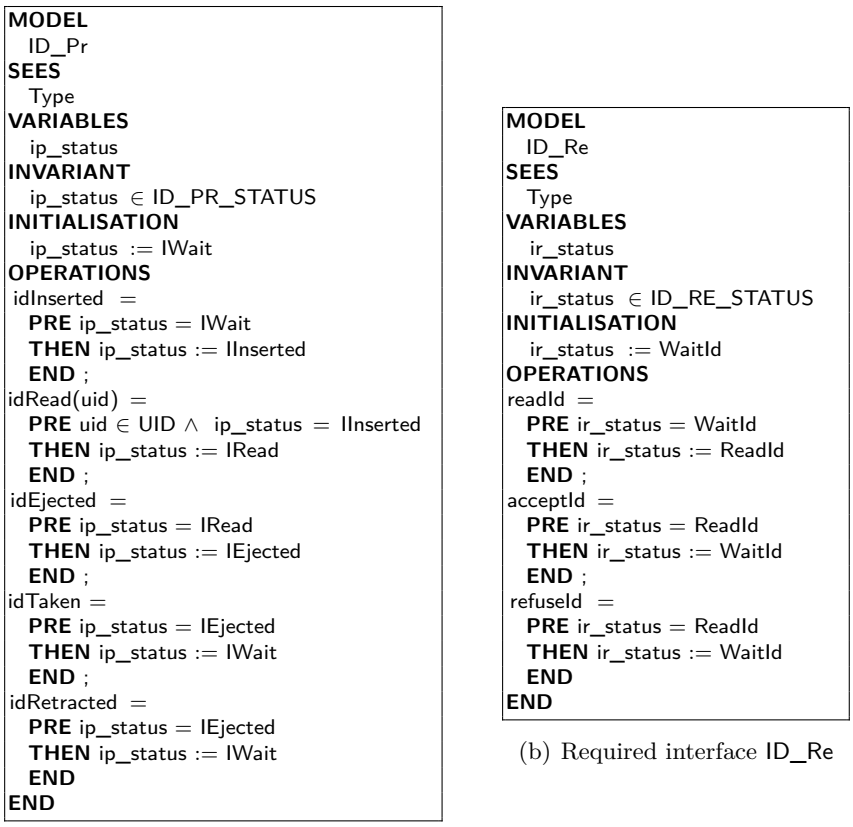
3. Component Trustworthy Assembly

Components must be assembled in an appropriate way. Interoperability means the ability of components to communicate and to cooperate despite differences in their implementation language, their execution environment, or their model abstraction [35]. Two components are interoperable if all their interfaces are compatible [11]. More precisely, it means that, for each required interface of a considered component, there exists a compatible interface which is provided by another existing component. Three main levels of interfaces compatibility are considered and checked:

- the syntactic level covers static aspects and concerns the interface signature. Each attribute of the required interface must have a counterpart in the provided one; for each method of the required interface, there exists an operation of the provided interface with the same signature,
- the semantic level covers behavioral aspects,
- the protocol level deals with the expression of functional properties (like the order in which a component expects its methods to be called).

A provided interface can propose more functionalities (attributes, methods, behaviors, protocols, etc.) than the required one needs, but all the functionalities used by the required interface must be proposed by the provided one. The process of proving interoperability between components is described in [11].

Often, to construct a working assembly out of components, adapters have to be defined, connecting the required interfaces to the provided ones. An adapter is a new component that realizes the required interface using the provided interface. At the signature level, it expresses the mapping between required and provided variables. At the behavioral and protocol levels, it expresses how the required operations are imple-



(a) Provided interface ID_Pr

(b) Required interface ID_Re

Figure 3. B Models for the interfaces of AccessControl

mented in terms of the provided ones. In [26], we have studied the adapter specification and its verification using B. We have given a B model of the adaptation that must refine the B model of the required interface including the provided incompatible interface.

More generally, the component assembly concerns the use of both types of interfaces for different components to be connected. We show that the component assembly is a generalization of the adaptation problem: a new specific component is introduced to manage the needed components. It realizes all the required interfaces of the considered components using their provided interfaces.

Example. Let us use a component Identification1 whose description is given in Figure 4 to answer the requirements of AccessControl presented Section 2. It is a card reader equipped with two lights, a green one and a red one. These lights indicate if the authorization has been accepted (the green light turns on) or denied (the red light turns on). The two lights cannot be turned on at the same time.

The component Identification1 is equipped with two interfaces:

- Its provided interface Ident_Pr, related to a system controller. Its two variables green and red give the state of two lights: the green light must be turned on if the authorization has been accepted (acceptIdent), otherwise the red light must be turned on (refuseIdent). An invariance property expresses that the two lights cannot be turned on at the same time, as expressed in the B model given in Figure 7a.
- Its required interface Ident_Re, related to a system controller which is similar to the ID_Pr provided interface of the AccessControl component previously defined.

To ensure that the assembly of components Identification1 and AccessControl is trustworthy

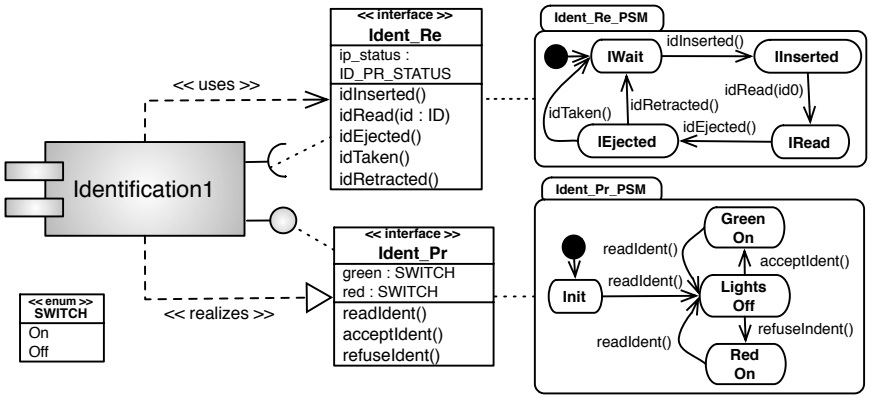


Figure 4. An existing component Identification1

[26], we must prove that the corresponding interfaces are compatible, as seen in Figure 5. We decompose this proof into two steps.

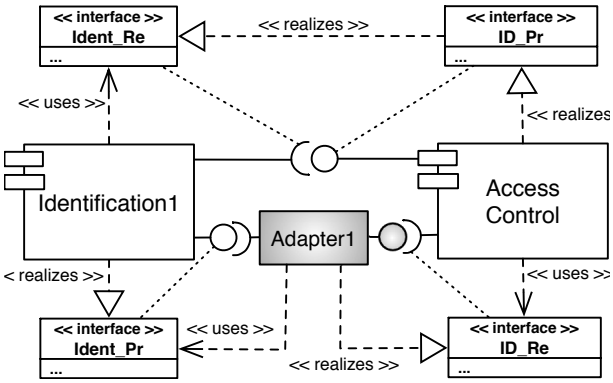


Figure 5. Identification1 and AccessControl assembly

3.1. Compatibility between Ident_Re and ID_Pr

We have to prove that ID_Pr *realizes* Ident_Re. This property can be expressed by the B refinement concept. We show that the B model of ID_Pr is a correct refinement of the B model of Ident_Re. That means that the methods of the provided interface implement directly the methods of the required interface. In this example, the proof of this refinement is obvious.

We conclude that the required interface Ident_Re of the component Identification1 is compatible with the provided interface ID_Pr of the AccessControl component. The interoperability is verified at the signature, semantic and protocol levels.

3.2. Adaptation between ID_Re and Ident_Pr

When looking at the required interface ID_Re of the component AccessControl, as expressed in Figure 2, it is obvious that it is not directly compatible with the interface Ident_Pr of the component Identification1, as shown in Figure 4. We propose to introduce a new component, called Adapter1, to map correctly these two incompatible interfaces. In terms of UML, this new component *realizes* ID_Re, *using* Ident_Pr.

The correctness of this adaptation can be proved using the B method. It is expressed by the schema presented Figure 6, in which Adapter1 is modeled by a refinement which:

– *refines* the B model of the required interface ID_Re and
– *includes* the B model of the provided interface Ident_Pr.

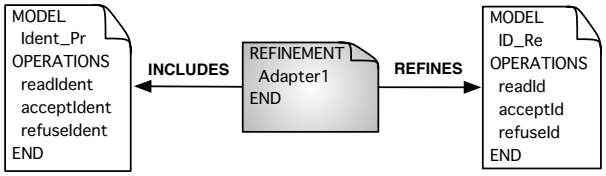
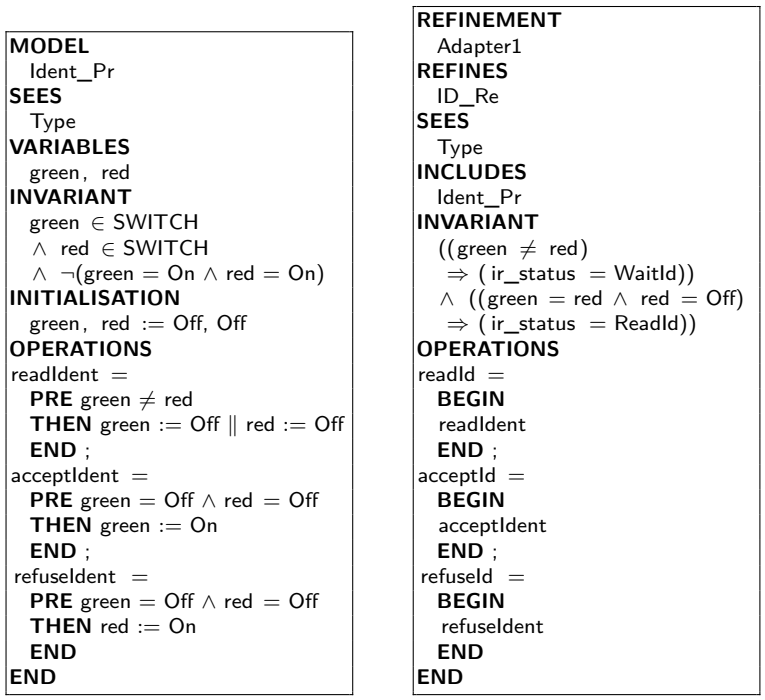


Figure 6. B adaptation between ID_Re and Ident_Pr

The B model of the component Adapter1 proposed in Figure 7b expresses the mapping between the two interfaces. The invariant clause, or gluing invariant, makes the correspondence between the required and the provided attributes.



```
MODEL
  Ident_Pr
SEES
  Type
VARIABLES
  green, red
INVARIANT
  green ∈ SWITCH
  ∧ red ∈ SWITCH
  ∧ ¬(green = On ∧ red = On)
INITIALISATION
  green, red := Off, Off
OPERATIONS
readIdent =
  PRE green ≠ red
  THEN green := Off || red := Off
  END ;
acceptIdent =
  PRE green = Off ∧ red = Off
  THEN green := On
  END ;
refuseIdent =
  PRE green = Off ∧ red = Off
  THEN red := On
  END
END
```

(a) B Model of Ident_Pr

```
REFINEMENT
  Adapter1
REFINES
  ID_Re
SEES
  Type
INCLUDES
  Ident_Pr
INVARIANT
  ((green ≠ red)
   ⇒ ( ir_status = WaitId))
  ∧ ((green = red ∧ red = Off)
   ⇒ ( ir_status = ReadId))
OPERATIONS
readId =
  BEGIN
  readIdent
  END ;
acceptId =
  BEGIN
  acceptIdent
  END ;
refuseId =
  BEGIN
  refuseIdent
  END
END
```

(b) B Model of Adapter1

Figure 7. B models for the adaptation

The variable ir_status required by the access control is defined in terms of the two variables green and red provided by the identification. The operation clause defines how the required methods, i.e. readId, acceptId and refuseId are implemented by the provided ones, readIdent, acceptIdent and refuseIdent.

To prove this refinement, B4free generates 12 obvious proof obligations and 5 proof obligations. As an example, we show in Figure 8 one of these proof obligations concerning the refinement of the method acceptId: we have to prove that red = Off (expressed by red$1 = SW_Off on B4free), using the listed hypotheses. This proof concerns the preservation of the invariant of Adapter1 by the precondition of the used method acceptIndent of Indent_Pr. The 5 proof olbigations are automatically discharged by B4free. As a consequence, Adapter1 implements ID_Re in terms of Ident_Pr. We are able to assemble Identification1 to AccessControl through Adapter1.

4. General Case of Component Trustworthy Assembly

Let us define an identification component corresponding to the previous requirements given section 3 in terms of three existing components, namely CardReader, Timer and MultiLights. The component CardReader is used to read the authentication informations on an access card and the component Timer to indicate the time limit. Green and red lights are provided by the component MultiLights.

The general case of component assembly concerns the use of both type of interfaces for different components to be connected. A new specific component is defined to manage these components. It realizes all the required interfaces of the considered components using their provided interfaces.

4.1. Existing components

The functionality of each component is known by its interface descriptions presented below as

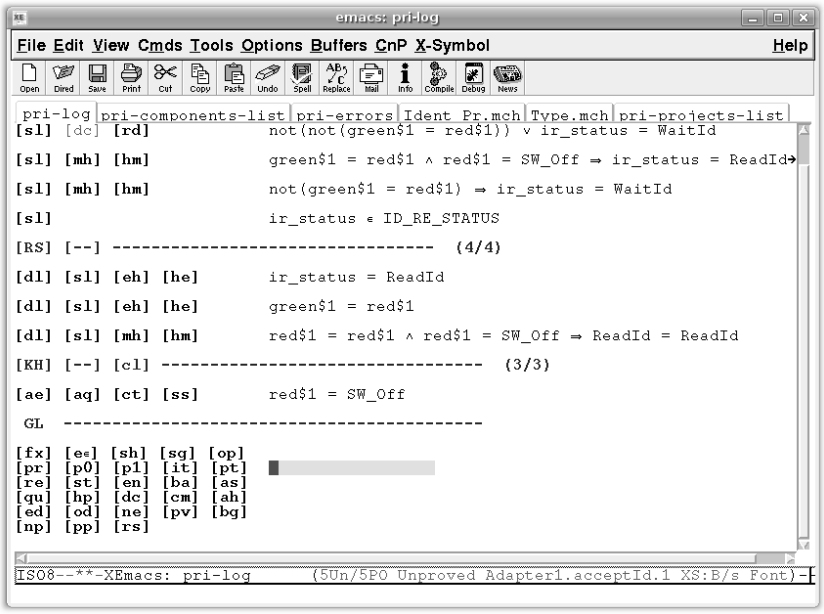


Figure 8. One of the proof obligations of the refinement of acceptId

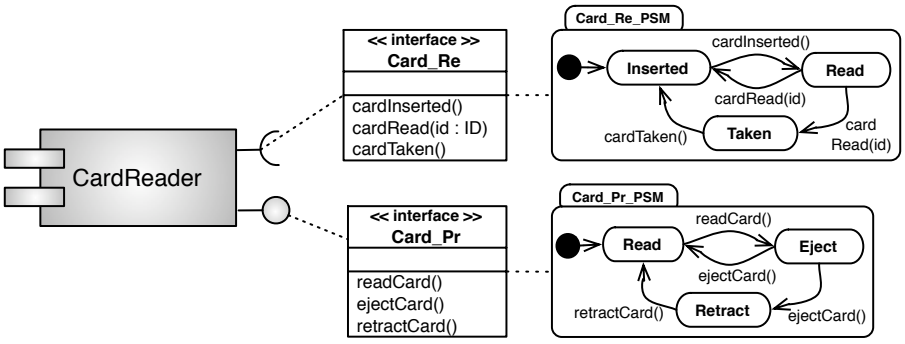


Figure 9. Component CardReader and its interfaces Card_Pr and Card_Re

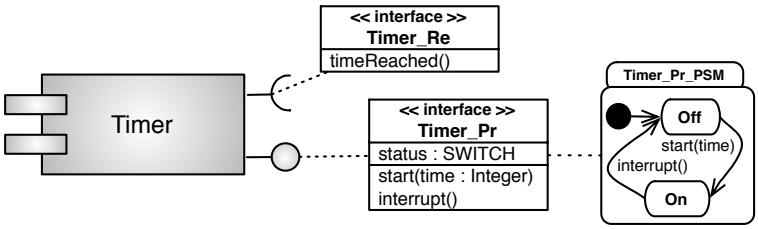


Figure 10. Component Timer and its interfaces Timer_Pr and Timer_Re

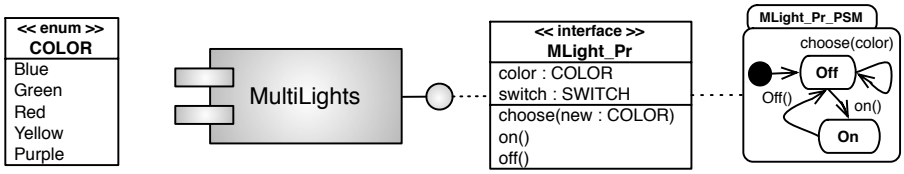


Figure 11. Component MultiLights and its provided interface MLight_Pr

UML 2.0 diagrams associated to B models for behavioral and protocol specifications.

The component CardReader**.** This component reads identification informations from an access card. It is equipped with two interfaces, as presented in Figure 9, a provided one named Card_Pr with three methods (readCard(), ejectCard() and retractCard()) and a required one named Card_Re which receives messages from its controller by the way of three methods.

The component Timer**.** As presented in Figure 10, this component has two interfaces. The provided one, Timer_Pr, offers two functionalities: it can be started with a fixed time and interrupted before the timeout is reached. When the timeout is reached, the timer sends this information through its required interface Timer_Re.

The component MultiLights**.** This component presented in Figure 11 is a light box that proposes several color lights. It offers, by the way of its provided interface MLight_Pr, the following functionalities: the chosen light can be turned on or turned off. When the light is turned off, one can choose a light color from predefined ones.

4.2. The component Identification2

A component Identification2 can be defined by assembling these three existing components, as depicted in Figure 12 in order to fulfill the requirements. The required and provided interfacesIdent_Re and Ident_Pr of Identification2 have to be defined in terms of the components CardReader, Timer and MultiLights through their interfaces Card_Re, Card_Pr, Timer_Re, Timer_Pr and MLight_Pr.

A "new" component Controller is introduced to manage the interactions between all these interfaces. Identification2 *delegates* to Controller its interfaces Ident_Re and Ident_Pr:

- Controller realizes for Identification2 the interface Ident_Pr, and
- Controller uses through Identification2 the interface Ident_Re.

Figure 13 shows the sequence of operation calls between all the components to be assembled to produce Identification2: there is the adaptation protocol between all the interfaces, that shows

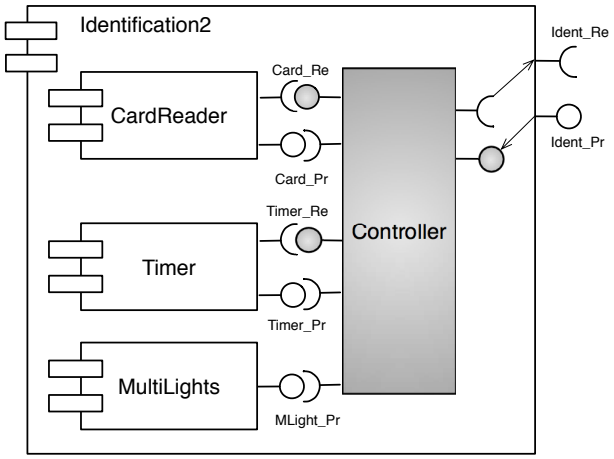


Figure 12. Architecture of the component Identification2

for all the required operations, the reaction in terms of provided operations calls.

To prove that the assembly of CardReader, Timer and MultiLights through Controller is correct, we must prove that Controller:

- *realizes* the provided interface Ident_Pr delegated by Identification2, *realizes* the required interface Card_Re of CardReader, and *realizes* the required interface Timer_Re of Timer,
- *uses* the provided interfaces Card_Pr, Timer_Pr and MLight_Pr of the three existing components, and the required interface Ident_Re delegated by Identification2.

This UML 2.0 architecture can be expressed by the B architecture given in Figure 14 with two levels of refinement:

- the B abstract model, Controller_abs, which *extends* all the interfaces to be realized,
- the B refinement model, Controller, which
  - *includes* all the interfaces to be used, and
  - *refines* the abstract model Controller_abs.

The B refinement model of Controller is given in Figure 15:

- the available components are included, i.e. Ident_Re, Card_Pr, Timer_Re and MLight_Pr
- its gluing invariant expresses how to obtain the required attributes green and red from the attributes color and switch of the provided interfaces,
- the operations clause describes all the needed methods in terms of the used ones. The sequence diagram, given in Figure 13, which

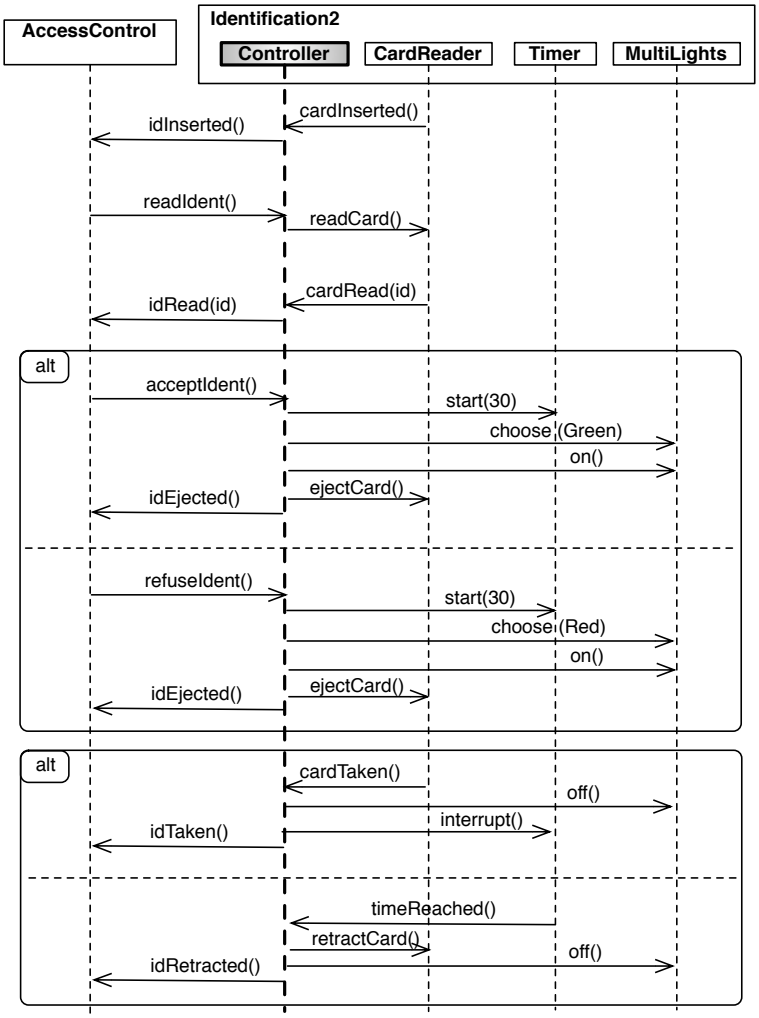


Figure 13. Sequence diagram for the Identification2 component

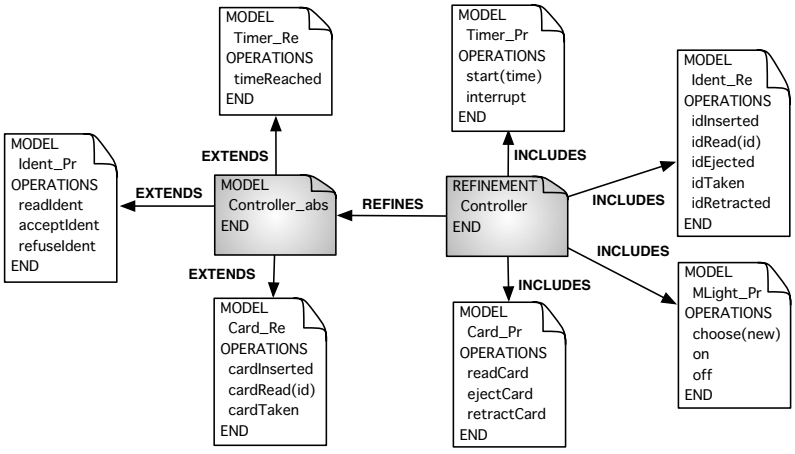


Figure 14. B architecture of the component Controller

gives the protocol of the adaptation can help us to express these needed methods.

As an example, let us consider the needed method acceptIdent() of the Ident_Pr interface of Identification2. This method is called when an inserted card has been authorized to enter the building. The required result, as expressed in the sequence diagram of Figure 13 must be that a green light is turned on during a fixed time and the card is ejected. This requirement is expressed in the B operation acceptIdent by:

1. a timer is started: start(30),
2. the light's color is fixed to green if it is necessary (method choose(green)) before the light is turned on (method on),
3. the card is ejected, ejectCard, and
4. the environment is informed, isEjected.

We prove using B4free that the proposed component Controller is a correct implementation of the required functionalities in terms of the three existing components. With the B prover, we check

– that Controller refines all the required interfaces. This guarantees that the required behavioral and protocol aspects are preserved by the assembling. Of course, the signature level is also considered,
– the correctness of the use of the provided interfaces by the inclusion of their B interface models.

The example of adapters presented in this paper is a part of the case study of an access control system, as presented in section 2. The AccessControl system is equipped with other interfaces not presented in this paper as shown in Figure 1: DB_Re, to be connected with a database and Entry_Pr, Entry_Re, and Exit_Pr, to manage turnstiles. Existing components are used to answer the requirements of the AccessControl system. The Database component provides an interface Database_Pr to access to the stored informations and the Turnstile component has two interfaces Turn_Pr and Turn_Re to lock and unlock the turnstile. The access control system has been completely developped using our component-based approach. AccessControl must be connected to other components through specific adapters. In this paper, we have presented Adapter1 and Controller. Other adapters Entry1, Entry2, Exit1 and Database1 (decomposed into three steps of refinement to ease the proof, giving three versions of the adapter) are similarly defined. The Table 1 gives an idea about all the proof obligations generated and discharged for the different components and adapters.

Table 1. POs of AccessControl

	Obvious POs	**POs**	**Interactives** POs
ID_Pr	11	0	0
ID_Re	7	0	0
Entry_Re	5	0	0
Exit_Pr	3	0	0
DB_Re	12	10	4
Ident_Pr	13	1	0
Ident_Re	11	0	0
Card_Pr	6	1	0
Card_Re	6	1	0
Timer_Pr	5	0	0
Timer_Re	0	0	0
MLight_Pr	11	0	0
Turn_Pr	5	0	0
Turn_Re	3	0	0
Database_Pr	3	0	0
Adapter1	12	5	0
Controller_abs	10	0	0
Controller	45	6	2
Entry1	9	2	0
Entry2	3	0	0
Database1-1	6	2	2
Database1-2	6	2	2
Database1-3	5	8	2
TOTAL	**203**	**38**	**12**

5. Related Works

In an earlier paper [19], we have investigated the necessary ingredients a component specification must have in order to be useful for assembly of a software system out of components. These ingredients are independent of concrete component models. We have proposed a method consisting of four steps to guide this process.

Several proposals for verifying the interoperability between components have been made. In [15], Estevez and Fillottrani analyze how to apply algebraic specifications with refinement to

```
REFINEMENT
   Controller
REFINES
   Controller_abs
SEES
   Type
INCLUDES
   Ident_Re, Card_Pr, Timer_Pr, MLight_Pr
INVARIANT
   (( color = Green ∧ switch = SW_On) ⇒ green = SW_On)
   ∧ (¬(color = Green ∧ switch = SW_On) ⇒ green = SW_Off)
   ∧ (( color = Red ∧ switch = SW_On) ⇒ red = SW_On)
   ∧ (¬(color = Red ∧ switch = SW_On) ⇒ red = SW_Off)
OPERATIONS
/* Ident_Pr */
readIdent =
   BEGIN
   readCard
   END ;
acceptIdent =
   BEGIN
   start (30) ;
   IF color = Green THEN on
   ELSE choose(Green) ; on
   END ;
   ejectCard ; idEjected
   END ;
refuseIdent =
   BEGIN
   start (30) ;
   IF color = Red THEN on
   ELSE choose(Red) ; on
   END ;
   ejectCard ; idEjected
   END ;
/* Timer_Re */
timeReached =
   BEGIN
   retractCard ; off ; idRetracted
   END ;
/* Card_Re */
cardInserted =
   BEGIN
   idInserted
   END ;
cardRead(uid) =
   BEGIN
   idRead(uid)
   END ;
cardTaken =
   BEGIN
   off ; interrupt ; idTaken
   END
END
```

Figure 15. B Model of the component Controller

component development, with a restriction to the use of modules that are described as class expressions in a formal specification language. They present several refinement steps for component development, introducing in each one design decisions and implementation details.

Our work focuses on the verification of interoperability of components through their interfaces using B assembling and refinement mechanisms.

Zaremski and Wing [36] propose an approach to compare two software components. They determine whether one required component can be substituted by another one. They use formal specifications to model the behavior of components and exploit the Larch prover to verify the specification matching of components.

In [9], a subset of the polyadic π-calculus is used to deal with the component interoperability at the protocol level. π-calculus is a well suited language for describing component interactions. Its main limitation is the low-level description of the used language and its minimalistic semantic. In [17, 18], protocols are specified using a temporal logic based approach, which leads to a rich specification for component interfaces. Henzinger and Alfaro [3] propose an approach allowing the verification of interfaces interoperability based on automata and game theories: this approach is well suited for checking the interface compatibility at the protocol level. In [6], the three levels of interface compatibilities are considered on web service interfaces described by transition systems.

Several proposals for component adaptation have already been made. The need of adaptation and assembly mechanisms was recognized in the late nineties [8, 14, 20]).

Some practice-oriented studies have been devoted to analyze different issues when one is faced with the adaptation of a third-party component [16]. A formal foundation to the notion of interoperability and component adaptation was set up in [35]. Component behavior specifications are given by finite state machines which are well known and support simple and efficient verification techniques for the protocol compatibility. Braccalia & al. [7] specify an adapter as a set of correspondences between methods and parameters of the required and provided components. An adapter is formalized as a set of properties expressed in π-calculus. From this specification and from both interfaces, they generate a concrete implementable adapter.

Reussner and Schmit consider a certain class of protocol interoperability problems in the context of concurrent systems. For bridging component protocol incompatibilities, they generate adapters using interfaces described by finite state machines [29, 30].

Automatic generation of adapters is limited as one has to ensure the decidability of the interfaces inclusion problem, which is necessary to perform automated interoperability checks; one could only generate adapters for specific classes of assembly.

In our approach, we are not only concerned with specific classes of interoperability but with adapters in general. We propose to give general schemes to specify and verify adapters, not to generate them automatically.

6. Conclusion and Perspectives

The success of the component-based paradigm has received considerable attention in the software development field in industry and academia like in other engineering domains. We have presented an approach which contributes to specify component-based systems with high safety requirements. Our approach concerns the first steps of the system development life-cycle, from the requirements phase to the specification one, and aims at using existing languages and tools. We focus on the integration of components and the assembly mechanisms: components are considered as black-boxes described by their visible behavior and their required and provided interfaces. To construct a working system out of existing components, adapters are introduced. An adapter is a piece of glue code that expresses the mapping between required and provided variables and how required methods are implemented in terms of the provided ones. We have presented a general schema of assembly with both cases of interfaces.

The use of the B formal method and its refinement and assembling mechanisms to model the component interfaces and the adapters allows us to define rigorously the interoperability between components and to check it with support tools. The B prover guarantees that the adapter is a correct implementation of the required functionalities in terms of the existing components. Within this approach, the verification of the interoperability between the connected components is done at three levels, the signature, the semantic and the protocol levels.

We are currently working on extending this approach when additional abnormal behaviors are introduced to increase dependability of our component architecture and to preserve the normal cases. We have introduced two kinds of dependability mechanisms, one for security and one for safety. To extend the proposed approach, we study the definition of adapter schemas corresponding to different cases of component architecture. The idea is not to automatically generate the adapters, but to propose schemas to develop and verify the adaptation.

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] AFADL'2000. Etude de cas : système de contrôle d'accès. In *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. actes LSR/IMAG.

[3] L. Alfaro and T.A. Henzinger. Interface automata. In *9th Annual Aymposium on Foundations of Software Engineering, FSE*, pages 109–120. ACM Press, 2001.

[4] P. Behm, P. Benoit, and J.M. Meynadier. METEOR: A Successful Application of B in a Large Project. In *Integrated Formal Methods, IFM99*, volume 1708 of *LNCS*, pages 369–387. Springer Verlag, 1999.

[5] D. Bert, S. Boulmé, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In *Integrated Formal Method, IFM '03*, volume 2805 of *LNCS*, pages 94–113. Springer Verlag, 2003.

[6] D. Beyer, A. Chakrabarti, and T.A. Henzinger. Web service interfaces. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, pages 148–159. ACM Press, 2005.

[7] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. In *Journal of Systems and Software*, volume 74, pages 45–54. Elsevier Science Inc., 2005.

[8] A.W. Brown and K.C. Wallnan. Engineering of component-based systems. In *Proceedings of*

the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), page 414. IEEE Computer Society, 1996.

[9] C. Canal, L. Fuentes, E. Pimentel, J-M. Troya, and A. Vallecillo. Extending CORBA interfaces with protocols. *Computer Journal*, 44(5):448–462, 2001.

[10] C. Canal, J.M. Murillo, and P Poizat. Software adaptation. *L'Objet*, 12(1):9–31, 2006.

[11] S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 160:157–172, 2006.

[12] S. Chouali and J. Souquières. Verifying the compatibility of component interfaces using the B formal method. In *International Conference on Software Engineering Research and Practice (SERP'05)*, pages 850–856. CSREA Press, 2005.

[13] Clearsy. B4free, 2004. `http://www.b4free.com`.

[14] I. Crnkovic, S. Larsson, and M. Chaudron. Component-based development process and component lifecycle. In *27th International Conference Information Technology Interfaces (ITI)*. IEEE, 2005.

[15] E. Estevez and P. Fillottrani. Algebraic Specifications and Refinement for Component-Based Development using RAISE. *Journal of Computer Science and Technologie*, 2(7):28–33, 2002.

[16] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6):17–26, 1999.

[17] J. Han. A comprehensive interface definition framework for software components. In *The 1998 Asia Pacific software engineering conference*, pages 110–117. IEEE Computer Society, 1998.

[18] J. Han. Temporal logic based specification of component interaction protocols. In *Proceedings of the Second Workshop on Object Interoperability ECOOP'2000*, pages 12–16. Springer-Verlag, 2000.

[19] D. Hatebur, M. Heisel, and J. Souquières. A Method for Component-Based Software and System Development. In *Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced Applications*, pages 72–80. IEEE Computer Society, 2006.

[20] G. Heineman and H. Ohlenbusch. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.

[21] M. Heisel, T. Santen, and J. Souquières. Toward a formal model of software components. In *Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02*, number 2495 in LNCS, pages 57–68. Springer-Verlag, 2002.

[22] H. Ledang and J. Souquières. Modeling class operations in B: application to UML behavioral diagrams. In *ASE'2001: 16th IEEE International Conference on Automated Software Engineering*, pages 289–296. IEEE Computer Society, 2001.

[23] H. Ledang and J. Souquières. Contributions for modelling UML state-charts in B. In *Third International Conference on Integrated Formal Methods – IFM'2002*, pages 109–127, Turku, Finland, 2002.

[24] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proceedings of the Formal Method Conference*, number 1708 in LNCS, pages 875–895. Springer-Verlag, 1999.

[25] Microsoft. *.Net*. `http://www.microsoft.com/net`.

[26] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation: Specification and Verification. In *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP'06), satellite workshop of ECOOP 2006*, pages 23–30, 2006.

[27] Object Management Group (OMG). *UML Superstructure Specification*, 2005. version 2.0.

[28] Object Mangagement Group (OMG). *Corba Component Model Specification*, 2006. version 4.0.

[29] R.H. Reussner, H.W. Schmidt, and I.H. Poernomo. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, pages 287–325, 2003.

[30] H.W. Schmidt and R.H. Reussner. Generating adapters fo concurrent component protocol synchronisation. In I. Crnkovic, S. Larsson, and J. Stafford, editors, *Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 213–229, 2002.

[31] Steria – Technologies de l'information. *Obligations de preuve: Manuel de référence, version 3.0*, 1998.

[32] Sun Microsystems. *JSR 220: Enterprise JavaBeans*, 2006. Version 3.0, Final Realase.

[33] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.

[34] W3C. *Web Services*. `http://www.w3.org/2002/ws`.

[35] D.D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[36] A.M. Zaremski and J.M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4):333–369, 1997.

e-Informatica Software Engineering Journal, Volume 2, Issue 1, 2008

Computation Independent Representation of the Problem Domain in MDA

Janis Osis*, Erika Asnina*, Andrejs Grave*

*Faculty of Computer Science and Information Technology, Institute of Applied Computer Systems, Riga Technical University

janis.osis@cs.rtu.lv, erika.asnina@cs.rtu.lv, andrejs.grave@cs.rtu.lv

Abstract

The object-oriented analysis suggests semiformal use-case driven techniques for problem domain modeling from a computation independent viewpoint. The proposed approach called Topological Functioning Modeling for Model Driven Architecture (TFMfMDA) increases the degree of formalization. It uses formal mathematical foundations of Topological Functioning Model (TFM). TFMfMDA introduces more formal analysis of the problem domain, enables defining what the client needs, verifying textual functional requirements, and checking missing requirements in conformity with the domain model. A use case model of the application to be build is defined from the TFM using a goal-based method. Graph transformation from the TFM to a conceptual model enables definition of domain concepts and their interrelation. This paper also outlines requirements to the tool to support TFMfMDA.

1. Introduction

The purpose of this work is to introduce more formalism into the problem domain modeling within OMG *Model Driven Architecture*® (*MDA*®) [19] in object-oriented software development. The main idea is to introduce a more formal definition of consistency between real world phenomena and an application that will work within these phenomena without introducing complex, hard to understand mathematics used while composing *Computation Independent Models (CIMs)*. For that purpose, formalism of a *Topological Functioning Model (TFM)* is used [22]. A TFM provides a **holistical** representation of system's **complete functionality** from the computation-independent viewpoint.

This paper is organized as follows. Section 2 describes related work. Section 3 describes key principles of *MDA*, and discusses suggested solutions of computation independent modeling and their weaknesses in the object-oriented analysis within MDA. Section 4 discusses a developed approach, i.e. *Topological Functioning Modeling for Model Driven Architecture (TFMfMDA)*, that makes it possible to use a formal model, i.e. a TFM, as a computation independent one without introducing complex mathematics. Besides that, it allows verifying of functional requirements at the beginning of analysis. TFMfMDA is illustrated by an application example in Section 5. Section 6 shows TFMfMDA conformity to the MDA Foundation Model. Section 7 describes requirements to the tool that should partially support automation of TFMfMDA. Conclusions state further directions of the research.

2. Related Work

Our work completely supports Jackson's work, which states that "...the principal parts of a software development problem are the machine, the problem world, and the requirements..." [15].

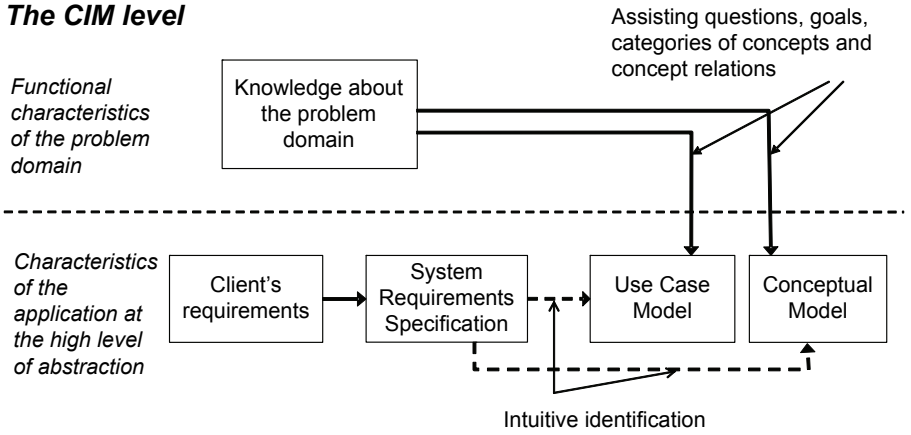


Figure 1. The current state of creation of the CIM in OOA

We also assume that the first step in the requirements gathering should be analysis of the "problem world" or "business" [10]. Therefore within TFMfMDA, the TFM describes functionality of the "problem world", while requirements describe functionality of the solution.

Analysis of the "business" context is also understood in goal-oriented requirements gathering approaches. Unfortunately, most of them are solution-orientated. Successful exceptions are KAOS methodology that analyzes the "problem world" and deals with conflicts by global representation of goals and agents [7], the $i*$ modeling framework that investigates agents that are assumed to be strategic and whose intentionality are only partially revealed [24], and, in some degree, the Requirement Abstraction Model [13] that links product requirements to organization's strategies. However, all these approaches operate rather with organization's strategic goals than with organization's functionality.

3. Construction of the CIM within MDA

Within MDA, the CIM usually includes several distinct models that describe system requirements, business processes and objects, an environment the system will work within, etc. Object-oriented analysis (OOA) is a semiformal specification technique that contains three steps: a) use case modeling, b) class modeling, and c) dynamic modeling. Use case usage is not systematic in comparison with systematic approaches that enable identifying of system requirement majority. Creation of use case models and determination of concepts and concept relations usually are rather *informal* than semiformal. Figure 1 shows several of the existing approaches of creating the mentioned models. Some approaches apply assisting questions [16, 18], category lists of concepts and concept relations (or noun-verb analysis) [17], or goals [6, 18] in order to identify use cases and concepts from the description of the system (in the form of informal description, expert interviewing, etc.). Other approaches draft a system requirements specification using classical requirements gathering techniques. Then these requirements are used for identification of use cases and creation of conceptual models. The most complete way is identification of use cases and concepts having knowledge of the problem world as well as a system requirements specification [2].

Use case modeling starts with some initial estimation (a tentative idea) about where the system boundary lies. For example, in the Unified Process [2], use cases are driven by requirements to the solution (but the business model is underestimated, and, thus, system boundaries are being identified intuitively), any requirement gathering technique can be applied, and requirements traceability to use cases is *ad hoc* defined. The B.O.O.M. approach [23] uses business-scope and system-scope use cases to make the solution more consistent with the problem world. The business-scope

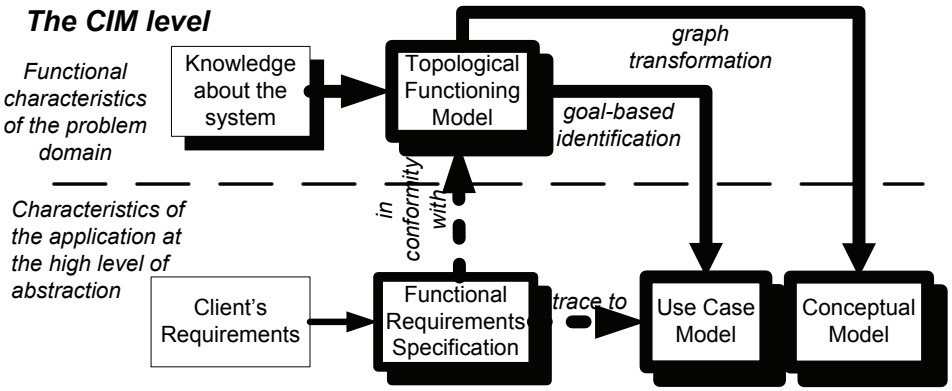


Figure 2. Creation of the CIM using TFMfMDA

use cases are used as a requirements gathering technique. Unfortunately, they are IT project driven not business driven. This means that analysis of the existing and planned business logic is also solution-oriented. Besides that, the traceability between system-scope use cases and business-scope use cases is captured with use-case packages that have their bottlenecks (intuitive and ad hoc creation; changes in business processes cannot be traceable in a natural way, etc.). Alistair Cockburn's approach [6] structures use cases with goals at different abstraction levels: system scope, goal specification, and interaction details. Despite benefits of such structuring, this approach also does not have proper problem domain analysis, and the multilevel character of the technique is not easy for everyone.

This means that the priority of problem domain modeling is very low. Thus, system functioning and its structure are based on intuitive understanding of the environment the system will work within. Until now use cases relate to the narrow area, where the real world interacts directly with the system (*the solution*), and, hence, focuses requirement analyst's attention on events that happen within *the solution* boundaries, but the properties of the surrounding real world can remain underestimated, e.g., software system requirements can conflict with rules that exist in the organization. Besides that, fragmentary nature of use cases does not give any answer on questions about: a) identifying all of the use cases for the system; b) conflicts among use cases; c) gaps that can be left in system requirements; d) how changes can affect behavior that other use cases describe [10, 11]. Use case checklists cannot completely help here, because reviews of lists of use cases are made only based on knowledge of the solution domain without formal connection to system's functionality in the problem world.

We consider that understanding and modeling the problem domain should be the primary stage in the software development, especially in case of embedded and complex business systems, which failure can lead to huge losses. This means that use cases must be applied as *a part of* a technique, whose first activity is construction of a well-defined problem domain model. Such an approach – *Topological Functioning Modeling for Model Driven Architecture (TFMfMDA)* is suggested in this paper. This research can be considered as a step towards MDA completeness and, therefore, towards MDA maturity.

4. Topological Functioning Modeling for MDA

This section discusses the proposed TFMfMDA approach. TFMfMDA main steps illustrated by bold lines in Figure 2 are discussed further in the paper. The approach is based on the formalism of a Topological Functioning Model and uses some capabilities of universal category logic [4, 3, 22].

As previously discussed, there are two interrelated branches at the beginning of system analysis: The first one is analysis of the problem world (the business or enterprise level), and the second one is analysis of the possible solution (the application level). Having knowledge about the

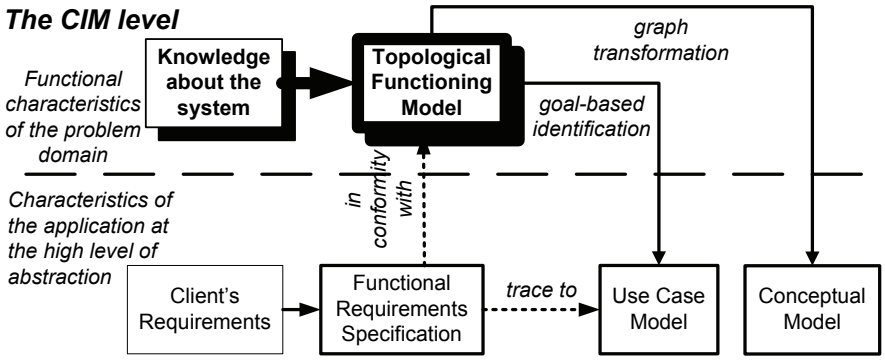


Figure 3. Construction of the TFM within TFMfMDA

complex system that operates in the real world, a topological functioning model of this system could be composed (Figure 2). This composed TFM is used to verify functional requirements and may be partially changed by them. TFM functional features are associated with business goals of the system; this provides identification of business-scope use cases as well as system-scope use cases in conformity with problem world's actualities. As a result, functional requirements are not only in conformity with the business-scope system's functionality but also can be traceable to the system-scope use case model. Problem domain concepts are selected and described in UML Class Diagram.

The TFM has a rigor mathematical base. It is represented in the form of topological space (X,Θ), where X is a finite set of functional features of the system under consideration, and Θ is the topology that satisfies axioms of topological structures and is represented in the form of a directed graph. "In combinatorial topology, the goal is to represent a topological space as an union of simple pieces. The word 'combinatorial' is used to suggest that the properties of the topological space rely on how the simple pieces are arranged. A graph is a simple combinatorial topological space." [5]. The necessary condition for construction of the topological space is a meaningful exhaustive verbal, graphical, or mathematical description of the system. The adequacy of the model describing functioning of a system can be achieved by analyzing mathematical properties of such an abstract object [22].

A TFM has as topological properties, namely, *connectedness, closure, neighborhood*, and *continuous mapping*, as functional properties, namely, *cause–effect relations, cycle structure, inputs* and *outputs*. These properties set model capabilities such as formal separation of subsystems, formal abstraction and refinement of the TFM, and analysis of similarities and differences of functioning systems. The last point relates to the structure of cycles in the TFM. It is proved that every business and technical system is a subsystem of its environment. The common characteristic of functionality of all systems (technical, business, or biological) is a **main feedback circuit**, whose visualization is **an oriented cycle**. Therefore, topological modeling states that at least one directed closed loop must be in every topological model of system functioning. This cycle visualizes the "main" functionality that has vital importance to the system's life. Usually feedback is expressed as an expanded hierarchy of cycles. Therefore, proper analysis of cycles is mandatory in composing the TFM, because it supports careful analysis of system's operation and interaction with its environment [21]. Composition of the TFM is discussed in Section 4.1.

4.1. Construction of the Topological Functioning Model

This section discusses construction of the TFM that represents the problem world in business context (Figure 3). Its steps illustrated in Figure 4 are the following: a) Definition of physical or business functional characteristics, b) Introduction of the topology, and c) Separation of the TFM.

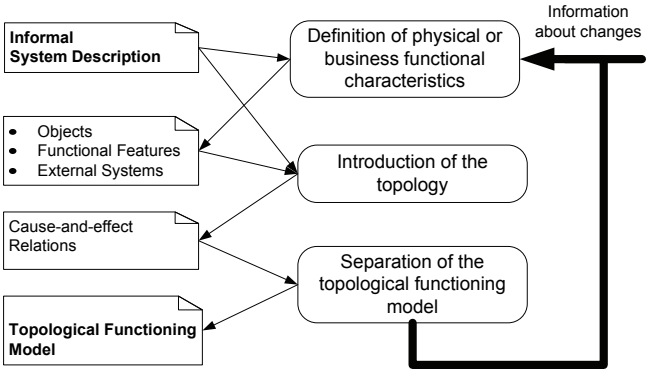


Figure 4. The method of construction of the TFM

Definition of physical or business functional characteristics consists of the following activities: 1) Definition of objects and their properties from the description of the problem world is performed by noun analysis, i.e. by establishing as meaningful nouns and their direct objects as handling synonyms and homonyms; 2) Identification of external systems (objects that are not subordinated to the system rules) and partially-dependent systems (objects that are partially subordinated to the system rules, e.g. workers' roles); and 3) Definition of functional features is performed by verb analysis, i.e. by founding meaningful verbs in the description. Each functional feature is a unique tuple *<A, R, O, PrCond, E>*, where A is an object action, R is a result of this action, O is an object (objects) that receives the result or that is used in this action (for example, a role, a time period, a catalog, etc.), *PrCond* is a set $PrCond = \{c_1 \dots c_i\}$, where c_i is a precondition or an atomic business rule (optional), and E is an entity responsible for action performing. Each precondition and atomic business rule must be either defined as a functional feature or assigned to the already defined functional feature. Two forms of textual descriptions are defined. The first is the more detailed form: *<action>-ing the <result> [to,into,in,by,of,from] a(n) <object>, [PrCond,] E*. An example is "*Check*-ing *out* the *availability* of a *copy*, PrCond = {a valid reader account}, E = a librarian". The latter is the more abstract form: *<action>-ing a(n) <object>, [PrCond,] E*. An example is "*Check*-ing *out* a *copy*, PrCond = {a copy is available}, E = a librarian".

Introduction of the topology Θ is the establishing of cause–effect relations between functional features. Cause-effect relations are represented as arcs of a digraph that are oriented from a cause vertex to an effect vertex. A structure of such relations can form a causal chain, wherein each relation is important.

Morevoer, cause–effect relations can form cycles. Therefore, cause–effect relations should be carefully checked whether they form **cycles** or **subcycles** in order to completely identify existing functionality of the system. The main cycle (cycles) of system functioning (i.e. functionality that is vitally necessary for system life) must be found and analyzed before starting further analysis. In case of studying a complex system, a TFM can be separated into a series of subsystems according to identified cycles.

Separation of the topological functioning model is performed by applying the closure operation over a set of system's inner functional features [22]. A *topological space* is a system represented by $Z = N \bigcup M$. Where N is a set of system's inner functional features, and M is a set of functional features of other systems interacting with the system or those of the system itself, which affect external systems. The TFM (X, Θ) is separated from the topological space of the problem world by the closure operation over the set N as it is shown by the equation

$$X = [N] = \bigcup_{\eta=1}^{n} X_\eta.$$

Where X_η is an adherence point of the set N and capacity of X is the number n of adherence points of N. An *adherence point* of the set N is a point, whose each neighborhood includes at least one point from the set N. The *neighborhood* of a vertex x in a digraph is the set of all vertices adjacent to x and the vertex x itself. It is assumed here that all vertices adjacent to x lie at the distance $d = 1$ from x on ends of output arcs from x. Moreover, a TFM can be separated into a series of subsystems by the closures of chosen subsets of N. The closure is illustrated in Section 5.

4.2. Functional Requirements Conformity to the TFM

The next step is verification of functional requirements (hereafter: requirements) whether they are in conformity with the constructed TFM. TFM functional features specify functionality that *exists in the problem world*, and functional requirements specify functionality that *must exist in the solution* [14]. Thus, it is possible to map requirements onto TFM functional features (Figure 5).

Mappings are specifyed using arrow predicates. An arrow predicate is a construct borrowed from the universal categorical logic. Universal categorical (arrow diagram) logic for computer science was explored in detail in Zinovy Diskin's et al. work [8].

Within TFMfMDA, five types of mappings together with corresponding arrow predicates are defined. **One to One.** *Inclusion predicate* (Figure 6a) is used if the requirement A completely specifies what will be implemented in accordance with the functional feature B. **Many to One.** *Covering predicate* (Figure 6b) is used if the requirements $A_1, A_2, \ldots, A_n$ overlap the specification of what will be implemented in accordance with the functional feature B. In case of the covering requirements, their specification should be precised. *Disjoint (component) predicate* (Figure 6c) is used if the requirements $A_1, A_2, \ldots, A_n$ together completely specify the functional feature B and do not overlap each other. **One to Many.** *Projection* (Figure 6d) is used if some part of the functional requirement A incompletely specifies the functional feature B_i. *Separating family of functions* (Figure 6e) is used if one requirement A completely specifies several functional features $B_1, \ldots, B_n$. It can be because: a) the requirement joins several ones and can be split up, or b) the functional features are more detailed than the requirement. **One to Zero.** One requirement specifies new or undefined functionality. In this particular case it is necessary to define possible changes of the problem domain's functioning (see Figure 4 "Information about changes"). **Zero to One.** The requirements specification does not contain any requirement related to the defined functional feature. This means that it can be a missed requirement and, hence, it could be not implemented in the application. Thus, it is mandatory to take a decision about implementation of the discovered functionality together with the client.

The result of this activity are both verified requirements and the TFM, which describes needed (and possible) functionality of the system and its environment.

4.3. Construction of the Use Case Model

The next step is transition from the model of the problem world constrained by the requirements to the use case model, supporting the possibility of more formal tracing of requirements to use cases (Figure 7).

This activity includes the following steps: a) Identification of system's users and their goals, b) Identification and refinement of system use cases, and c) Prioritization of use cases (and requirements).

Identification of system's users and their goals. At this stage, the TFM represents functionality of the problem world constrained by the requirements. System's users can be those, who interacts within the business system (workers) and with the business system (actors). *Actors* are external companies, clients, etc. *Workers* are system's *inner* entities (humans, roles, etc.) Identification of system users' *direct* **goals** is related to the identification of the corresponding set of functional features that are

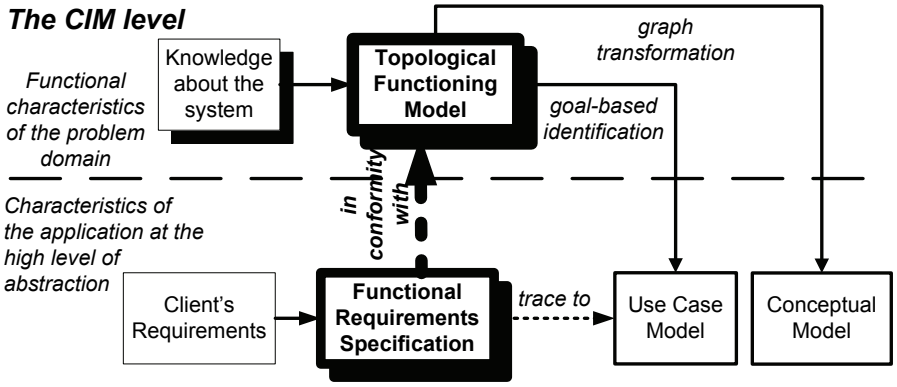


Figure 5. Making functional requirements in conformity with the TFM

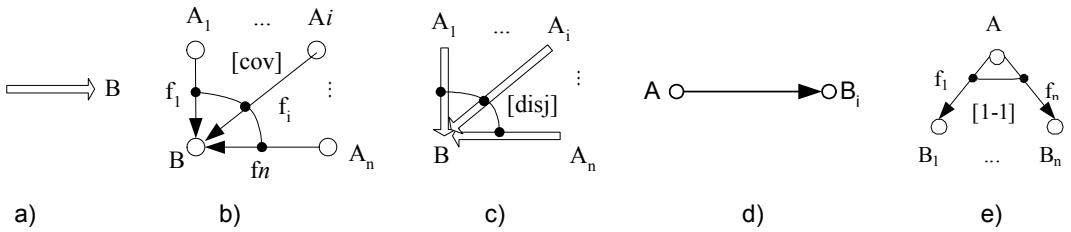


Figure 6. Functional requirements mapping onto TFM functional features

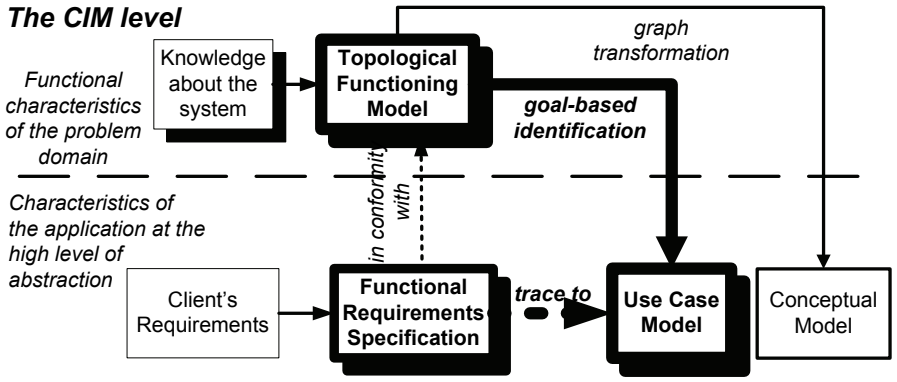


Figure 7. Construction of the use case model within TFMfMDA

necessary for satisfaction of these goals. A goal as a means for identification of use case has been chosen because it can be achieved performing some process that can be long running. The time gap cannot do this. For each goal, an input functional feature (input transaction), an output functional feature (output transaction), and a functional feature chain between them can be defined. Both actors and workers can be users of the application. Identification of system-scope goals helps in verifying additional requirements, e.g., for discovering "missing" requirements.

Identification and refinement of system use cases. Functional features mapped by functional requirements that are grouped together by a goal describe functionality necessary for ***achievement of this goal***, and, hence, describe ***a system-scope use case***. System's users that establish the goal are (UML) actors that communicates with such use cases. This principle enables formal identification of a use case model from the TFM. However, this principle provides also additional possibilities for refinement of the system use cases. An *inclusion use case* is some

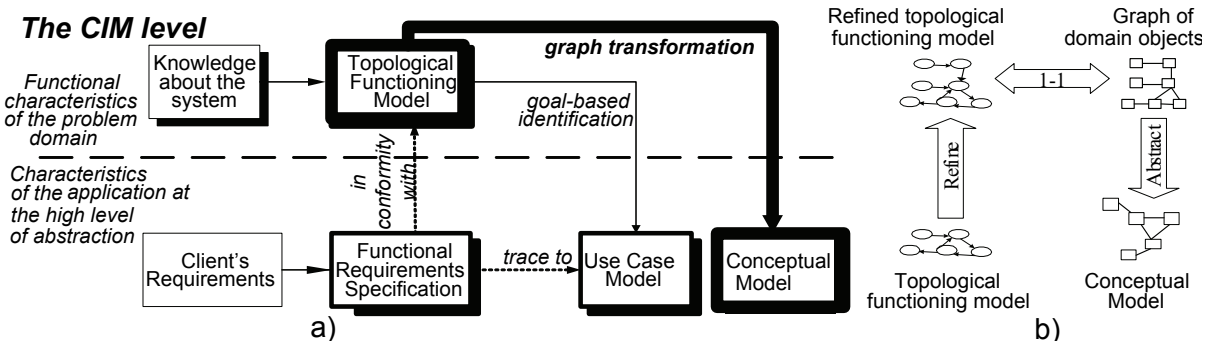


Figure 8. The step (a) and the process (b) of construction of the conceptual model

common sequence for several use cases. In the TFM, it is an intersection of sets of functional features that belongs to more than one system goals. Each common functional feature must be analyzed. The common functional feature in the main flow of a use case is a candidate to an inclusion use case. An *extension use case* shows an alternative way of the scenarios execution. In the TFM, it is functional features in a sub-cycle or a branch, existing within the system goal. The point of branch beginning is an extending point. Identified use cases can be represented in UML Activity Diagram by transforming functional features into diagram's activities, and cause–effect relations into diagram's control flows.

Prioritization of use cases. Prioritization of use cases and, thus, functional requirements can be done in accordance with client's desires or using requirements attribute systems, e.g. MoSCoW or GRASP [2]. Within TFMfMDA, priorities of implementation of use cases are defined in conformity with the TFM main cycle as follows (in accordance with the Rational Unified Process): a) *critical (must be implemented otherwise the application will not be acceptable)* – if a use case implements any functional feature that belongs to the main functional cycle; b) *important (it would significantly affect the usability of the application)* – if a use case implements any functional feature that is a cause or an effect of a functional feature that belongs to the main cycle; and c) *useful (it has a low impact on the acceptability of the application)* – if a use case does not implement any functional feature of the main cycle or functional feature that affects or is affected by a functional feature that belongs to the main cycle.

4.4. Construction of the Conceptual Model

The last step of TFMfMDA is identification of the conceptual model. After requirements mapping, the TFM represents functionality that must be implemented in the application, and includes all concepts that are necessary for proper system's functioning (Figure 8a).

In order to obtain a conceptual model, it is necessary to detail each TFM functional feature to the level when it describes only objects of one type. This more precise model must be transformed one-to-one into a graph of domain objects. Then vertices with objects of the same type must be merged keeping all cause–effect relationships to graph vertices, which contain objects of other types (this is illustrated by the example in Section 5). The result is a graph of domain objects with indirect associations (Figure 8b). In order to make these relations more precise, the graph can be transformed into a sketch [8], then refined, and represented as a refined conceptual model. This transformation also indicates possible inheritance relations among types, and common operations, which can further be transformed into use case interfaces.

5. An Example of Application

This section gives an example of apllying TFMfMDA. Let us consider the *small fragment of an informal description* of the system from the project, within which the application for a library was developed. In this fragment, nouns are denoted by *italic*, verbs are denoted by

bold, and action pre- (or post-) conditions are underlined.

"When an unregistered *person* **arrives**, the *librarian* **creates** a new *reader account* and a *reader card.* The *librarian* **gives out** the *card* to the *reader.* When the *reader* **completes** the *request for a book*, hi **gives** it to the *librarian.* The *librarian* **checks out** the requested *book* from a *book fund* to a *reader*, if the *book copy* is available in a *book fund.* When the *reader* **returns** the *book copy*, the *librarian* **takes** it **back** and **returns** the *book* to the *book fund.* He **imposes** the *fine* if the *term of the loan* is exceeded, the *book* is lost, or is damaged. When the *reader* **pays** the *fine*, the *librarian* **closes** the *fine.* If the *book copy* is hardly damaged, the *librarian* **completes** the *statement of utilization*, and **sends** the *book copy* to the *Utilizer*."

Construction of the TFM. The identified objects (or concepts) are the following: a) inner objects are a librarian (L), a book copy (a synonym is a book), a reader account, a reader card, a request for a book, a fine, a loan term, a statement of utilization, book fund, and b) external objects are a person (P), a reader (R), and an utilizer (U).

The identified functional features are represented as *<number: a description of the functional feature, a precondition, a repsonsible entity and subordination>*, where "In" denotes "inner", and "Ex" denotes "external" subordination. They are the following: 1: Arriving a person, {}, P, Ex; 2: Creating a reader account, {unregistered person}, L, In; 3: Creating a reader card, {}, L, In; 4: Giving out the reader card to a reader, {}, L, In; 5: Getting a reader status, {}, R, Ex; 6: Completing a request for a book, {}, R, In; 7: Sending a request for a book, {}, L, In; 8: Checking out the book copy from a book fund, {}, L, In; 9: Checking out the book copy to a reader, {completed request AND book copy is available}, L, In; 10: Giving out a book copy, {}, L, In; 11: Getting a book copy, {}, R, Ex; 12: Returning a book copy, {}, R, Ex; 13: Tacking back a book copy, {}, L, In; 14: Checking the term of loan of a book copy, {}, L, In; 15: Evaluating the condition of a book copy, {}, L, In; 16: Imposing a fine, {the loan term is exceeded OR the lost book OR the damaged book}, L, In; 17: Returning the book copy to a book fund, {}, L, In; 18: Paying a fine, {imposed fine}, R, In; 19: Closing a fine, {paid fine}, L, In; 20: Completing a statement of utilization, {hardly damaged book copy}, L, In; 21: Sending the book copy to Utilizer, {}, L, In; 22: Utilizing a book copy, {}, U, Ex.

In order to define system's functionality – the set X, we perform the closuring operation over the set of system's inner functional features N ={2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20}. The set of external functional features and system's functional features that affect the external systems M = {1, 4, 5, 18, 21, 22}. The neighborhood of each element of the set N is as follows: $X_2 = \{2, 3\}$, $X_3 = \{3, 4\}$, $X_6 = \{6, 7\}$, $X_7 = \{7, 17\}$, $X_8 = \{8, 9\}$, $X_9 = \{9, 10\}$, $X_{10} = \{10, 11\}$, $X_{11} = \{11, 5\}$, $X_{12} = \{12, 13\}$, $X_{13} = \{13, 14\}$, $X_{14} = \{14, 15, 16\}$, $X_{15} = \{15, 16, 17, 20\}$, $X_{16} = \{16, 19\}$, $X_{17} = \{17, 8\}$, $X_{19} = \{19\}$, $X_{20} = \{20, 21\}$. The obtained set is X = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21}.

The identified cause–effect relations between the functional features are illustrated in Figure 9a. The main functional cycle is defined by an expert and includes the following functional features "17-8-9-10-11-5-12-13-14-15-17". It is denoted by bold lines in Figure 9a. These functional features describe checking out and taking back a book. They are assumed to be main, because have a major impact on business system's operation. The example of the first order subcycle is "5-6-7-17-8-9-10-11-5".

Functional requirements conformity to the TFM. Let us assume that the drafted functional requirements (FR) are as follows. **FR1:** The system shall perform registration of a new reader; **FR2:** The system shall perform check out of a book copy; **FR3:** The system shall perform check in of a book copy; **FR4:** The system shall perform imposing of a fine to a reader; and **FR5:** The system shall perform handling of an unsatisfied request (the description: the unsatisfied request should be added to the wait list; when a book copy is returned to the book fund, the system checks

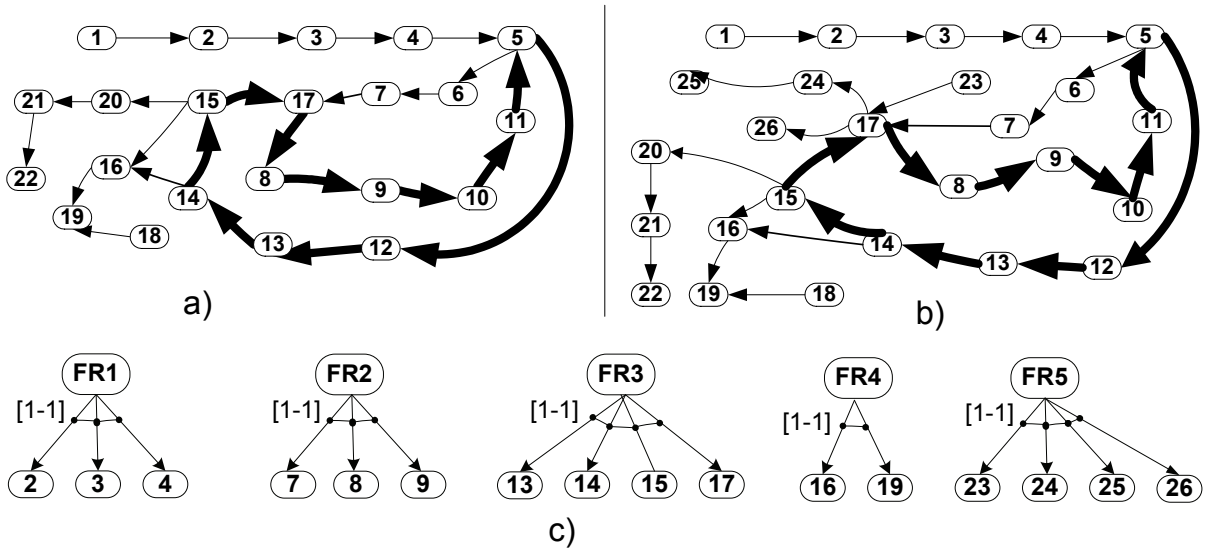


Figure 9. The topological space (a) and the modified topological space (b) of the library functioning; the correspondence between requirements and TFM functional features (c)

what request can be satisfied and, in success, informs the readers by SMS).

FR1 maps onto the functional features 2, 3, and 4, i.e. FR1 = {2, 3, 4}; FR2 = {7, 8, 9}, FR3 = {13, 14, 15, 17}, FR4 = {16}. The functional requirement FR5 describes new functionality that must be implemented in the application and introduced in the business activities of the system. System's functionality described in the TFM by the functional features 18, 19, 20, and 21 is not specified by requirements. This means that more careful analysis of the requirements and problem world is needed, because they can be missed. The better way in this situation is to specify these features in the requirements specification (and as use cases). The final decision must be taken together with the client that is warned beforehand about possible negative aftereffects. In this context, the interesting one is the functional feature 19, which describes closing of an imposed fine. It should be implemented. Therefore, FR4 is modified as *"The system shall perform imposing and closing of a fine to a reader"*. Hence, FR4 = {16, 19}.

The new functionality introduced by FR5 can be described by new identified objects (the system, a wait list and SMS), and the following functional features – **23:** Adding the request_for_a_book in a wait list, {unavailable book}, L, In; **24:** Checking the request_for_a_book in a wait list, {a book copy is returned to the book fund}, system, In; **25:** Informing the reader by SMS, {a request in the wait list can be satisfied}, system, In; **26:** Avoiding a request for a book, {book copy is not available}, system, In.

Introducing this functionality into the TFM, we must recheck all the existing cause–effect relations between the previously identified functional features taking into account possible changes in causes and effects. The set $N = \{2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 23, 24\}$. The set $M = \{1, 4, 5, 18, 21, 22, 25, 26\}$. After the closuring, the set $X = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 23, 24, 25, 26\}$. The result is the model represented in Figure 9b. The final correspondence between the functional features and requirements is illustrated in Figure 9c. All the identified mappings of the requirements onto the functional features have the type "one-to-many".

Construction of the use case model. In order to define use cases, system's users and their goals together with necessary functional features are identified. System's users (Librarian, and System) are transformed into UML actors, goal names into use case names, and functional features into steps of the corresponding use cases. The resulting use case model, functional features to be implemented, and implementation prior-

ities of use cases defined accordingly to TFM functioning cycles are illustrated in Figure 10a. Figure 10b shows how two of the use cases can be described in UML Activity Diagram using information from the TFM, where functional features are transformed into activities, but cause–effect relations into control flows.

Construction of the conceptual model. The step of the TFM refinement is skipped, because each functional feature takes a deal with objects of the only one type. Figure 11 shows transformation of the TFM to the graph of domain objects. Additionally, Figure 12a reflects this graph after the gluing all graph vertices that represent functional features with objects of the same types. This reflects the idea proposed in [20, 21, 22] that the holistic representation of the domain by means of the TFM enables identifying of all necessary domain concepts, and, even, enables defining their necessity for successful implementation of the system.

6. MOF-based Metamodel of TFMfMDA

In compliance with [1], the Foundation Model of MDA requires that a metamodel of each modeling language used within MDA must be defined in Meta Object Facility (MOF) terms for conformance purposes. Therefore, a metamodel of TFMfMDA concepts was defined as well as an UML profile for TFMfMDA [4].

The MOF is a core standard of MDA. Its architecture has four *metalevels*. They are named M3, M2, M1 and M0 [12]. Conceptually the level M3 is the MOF itself, i.e. a set of constructs used to define metamodels. M2 describes instances of constructs from M3. M1 includes instances of metamodel constructs from M2. Finally, the level M0 describes objects and data that are instances of elements from M1. TFMfMDA constructs are made in conformity with these metalevels as illustrated in Figure 12b. The metamodel for TFMfMDA is described at the level M2 [22]. These metamodel illustrated in Figure 13 specifies how TFMfMDA concepts related to each other.

A topological functioning model is an instance of the type *TFMTopologicalFunctioningModel* that includes at least two functional features of the type *TFMFunctionalFeature*. They can be united in functional feature sets (*TFMFunctionalFeatureSet*). This means that a functional feature represented in the TFM can visualize a functional feature set. One functional feature can contain only one set and one functional feature can belong only to the one set. A functional feature can be subordinated to a business system itself or to an external system (*Subordination*). Functional features can form functioning cycles (*TFMCycle*) of different order. Functional features are connected by cause–effect relations. A causal functional feature must have at least one effect. An effect functional feature must have at list one cause. Functional features are mapped by functional requirements (*TFMFunctionalRequirement*) via the correspondence (*TFMCorrespondance*). The correspondence is many to many in general. It can be complete or incomplete, overlapping or disjoint. Functional features can be associated with several goals (*TFMUserGoal*) that are established by direct users (*TFMUserRole*) of the business system. The users can be external entities that interact with the business system (*TFMBusinessActor*) or workers that interact within the business system (*TFMBusinessWorker*). A user goal can be specialized to a business goal (*TFMUserBusinessGoal*) and to a system goal (*TFMUserSystemGoal*). The latter includes functional features to be implemented. This means that it includes functionality that is specified in the functional requirements specification. A user goal and, thus, corresponding functional requirements, are associated with functioning cycles, whose order affect a benefit value (*Benefit*) of implementing requirements.

7. Requirements to the Tool to Support TFMfMDA

As previously mentioned, TFMfMDA introduces certain formalism into the problem domain modeling from the computation independent

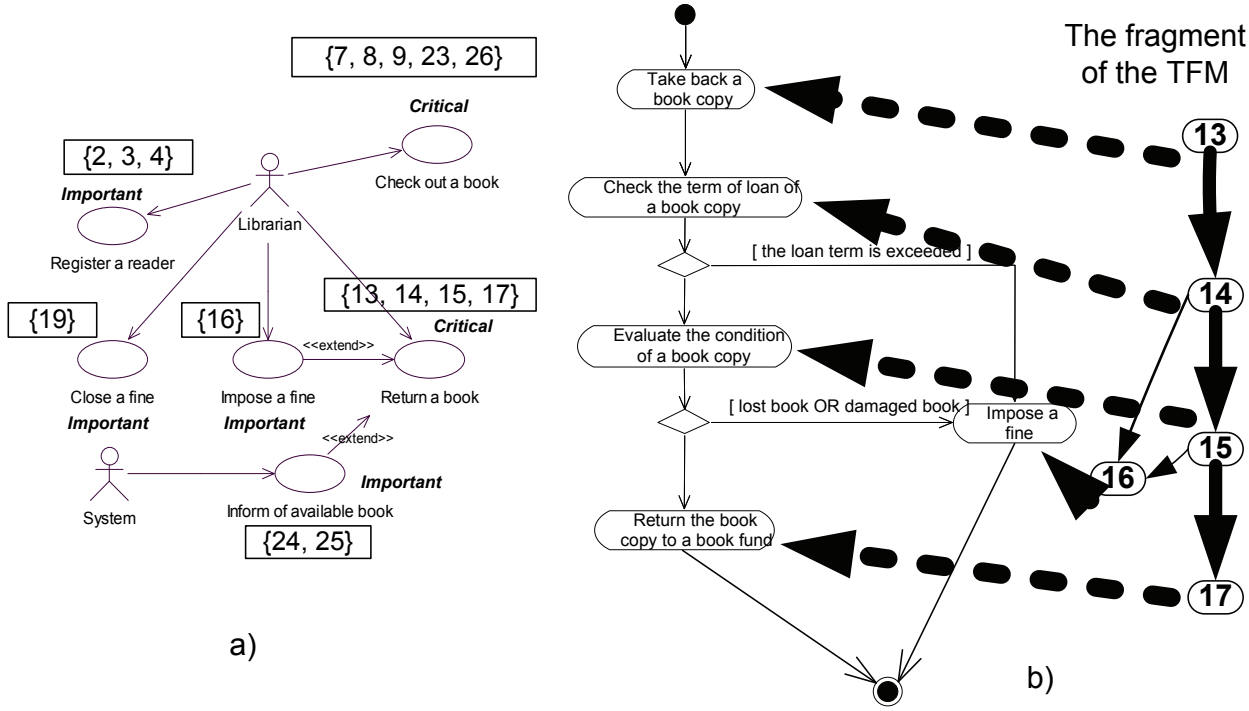


Figure 10. The use case model (a), and the fragment of the TFM described in UML Activity Diagram that specifies functionality of use cases "Return a book" and "Impose a fine" (b)

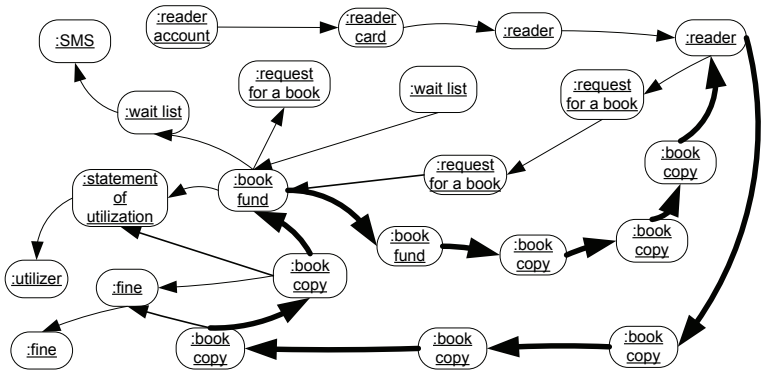


Figure 11. The graph of types of domain objects

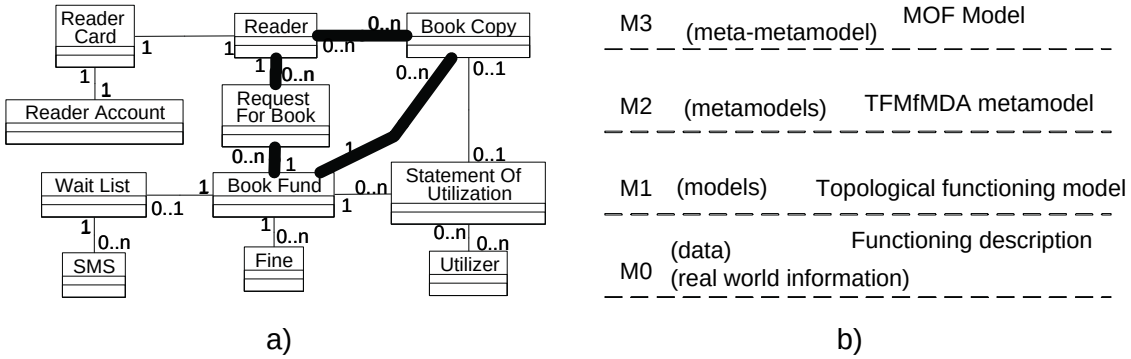


Figure 12. The initial conceptual model (a), TFMfMDA at the MOF metalevels (b)

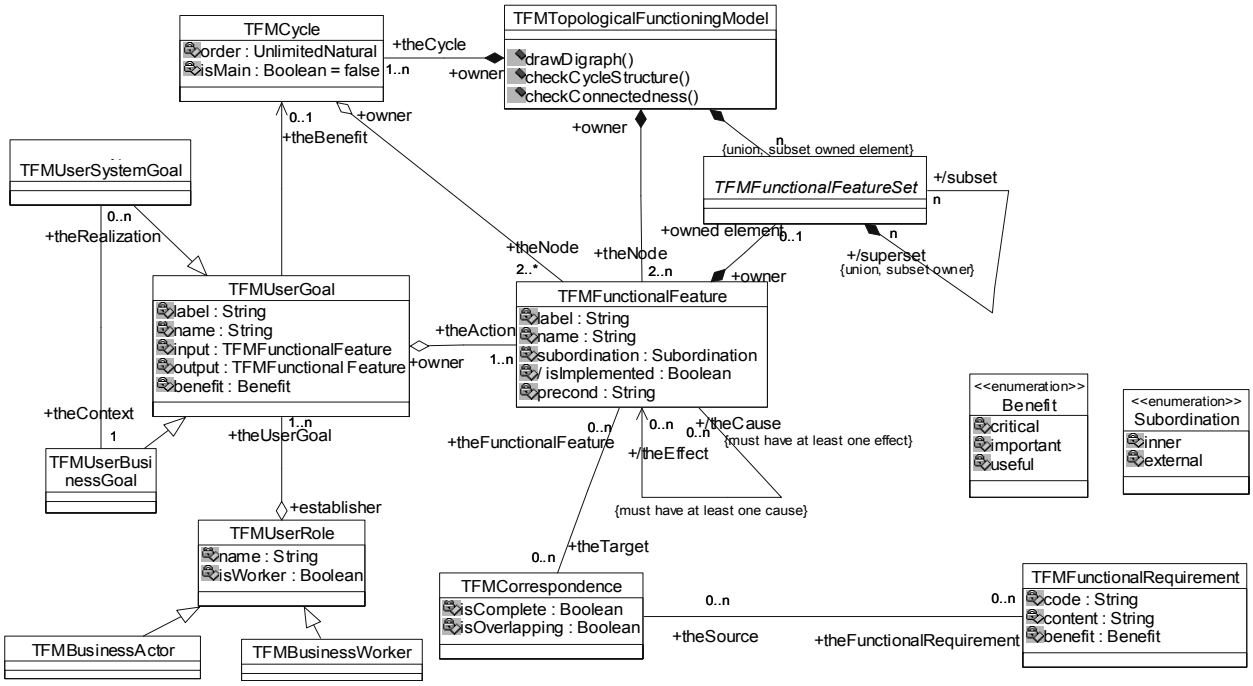


Figure 13. The MOF-based metamodel of TFMfMDA

viewpoint. Unfortunately, a use of complex graph-based constructs requires additional efforts. Therefore, the main purpose of the TFMfMDA tool is model management, which relates to model verification, traceability handling, automation of TFMfMDA steps, etc. This section discusses the requirements to the tool for TFMfMDA support.

The tool should support the client-server architecture. In case of the client-server architecture, the server should keep information of models; the client part should enable the connection with the server and use of the kept information. The tool should be realized as an Eclipse plug-in [9]. Eclipse is an open development platform that consists of different components, which helps in developing Integrated Development Environments (IDEs). For implementation of the tool the following Eclipse components can be used: Workbench UI, Help system, and Plug-in Development Environment (PDE). The Workbench UI is a component that is responsible for plug-in integration with Eclipse User Interface (UI). It defines extension points, using which a plug-in can communicate with the Eclipse UI. Help System is a component that provides complete integration of help information into the Eclipse help system. PDE is the environment that enables automation of activities related to the plug-in development.

The tool should enable work with textual information (an informal description of the system, a description of functional requirements) and graph-based constructs (a TFM, a conceptual model, and a use case model). All changes must be propagated automatically to all the related models. A general scheme of tool's activities is illustrated in Figure 14. The scheme describes TFMfMDA steps considered above in this paper. The first three steps reflects construction of the TFM. The fourth step reflects check of functional requirements and activities of enhancing the TFM. The fifth step illustrates creation of the use case model. Additionally, the sixth step shows composing of the conceptual model.

The challenge is realization of work with informal descriptions (Figure 15). The informal text should be handled on the server side because of several causes, namely, using of the knowledge base, the multi-user environment, and "learning" possibilities of the tool. The server side should support detection of nouns, noun phrases, and verbs. The detected information should be sent to

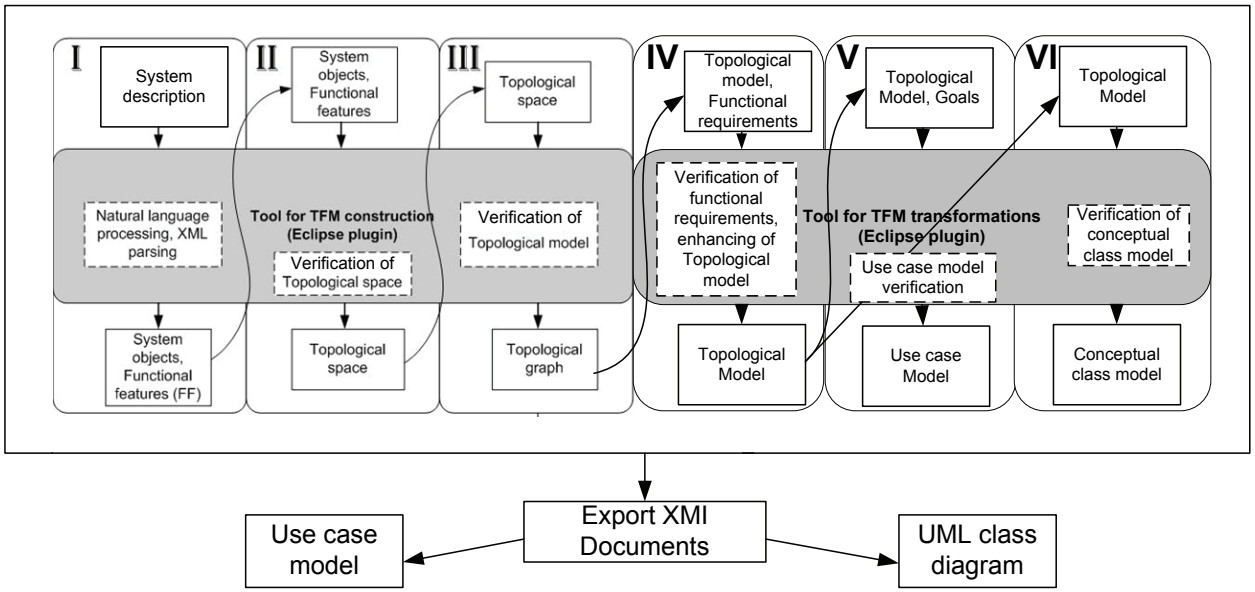


Figure 14. The general scheme of the tool supporting TFMfMDA

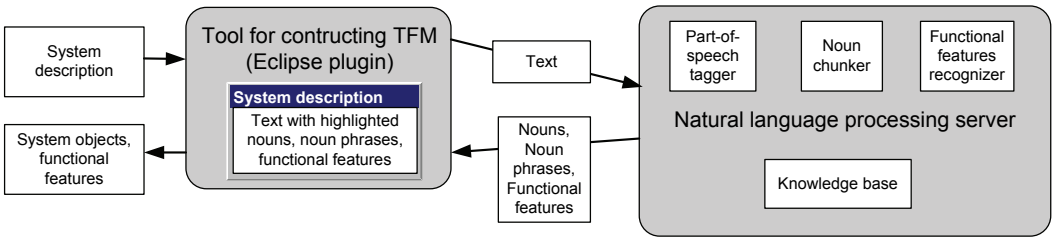


Figure 15. Handling the informal description of the system

the client side in XML file form. On the client side, it can be highlighted to the user in different ways (different colors, fonts, etc.). The tool must provide convenient interface for handling this information and creating TFM functional features.

Introduction of the topology between TFM functional features should be realized as a mix of graphical and textual representations of the functional features. The tool should offer a user to union or split up functional features, and to define cause–effect relations among them using tabular representations, but the result should be also represented in the graph form.

The TFMfMDA tool must provide a separate editor for each step. Each editor should have related views that help to represent information actual in this step for a user. All automated steps that require human participation should be realized as wizards.

8. Conclusions

The paper discusses about TFMfMDA and its application to certain formalism introducing in the process of creation of the CIM. TFMfMDA specifies complex systems using graph constructs and their transformations. Note that formal transformations of graphs are not limited with the number of vertices in graphs. The number of graph vertices can be decreased using formal abstraction of the graph. The primary goal of TFMfMDA is to specify functionality of the system in the problem domain. Certainly, the careful modeling of the problem domain requires additional expenses, but further it will be worthwhile, because it gives the formal CIM, decreases further expenses as decrease the number of development iterations, and facilitates change implementation.

TFMfMDA application has the following advantages. First, careful cycle analysis can help in identifying all (possible at that moment) functional and causal relations between objects in complex business systems. Implementation priorities of requirements can be set not only in accordance with client's whishes, but also in accordance with functioning cycles of the TFM. The latter makes it possible to take a decision about change acceptability in functionality of the problem domain before implementation of the changes in the application, and helps to check completeness of functional requirements. Second, TFMfMDA solves some use case limitations using formal mathematical means, e.g., it provides use case completeness, avoids conflicts among use cases, and shows their affect on each other. Besides that it does not limit a use of any requirements gathering techniques.

The tool built accordingly to the requirements would partially automate TFMfMDA steps described above. However, TFMfMDA requires human participation, thus, the further research is related to enhancing TFMfMDA with capabilities of natural language handling in order to make it possible to automate more steps of TFMfMDA and to decrease effect of human participation in decision making.

References

[1] A proposal for an MDA foundation model. ORMSC White Paper ormsc/05-04-01, OMG, `www.omg.org/docs/ormsc/05-04-01.pdf`, Apr. 2005. V00-02.

[2] J. Arlow and I. Neustadt. *UML2 and the Unified Process: Practical Object-Oriented Analysis and Design.* Addison-Wesley, Pearson Education, second edition, 2005.

[3] E. Asnina. Formalization aspects of problem domain modeling within model driven architecture. In O. Vasilecas, editor, *Databases and Information Systems. 7th International Baltic Conference on Databases and Information Systems. Communications, Materials of Doctoral Consortium*, pages 93–104, Vilnius, Lithuania, 2006. Vilnius Gediminas Technical University, Technika.

[4] E. Asnina. *Formalization of Problem Domain Modeling within Model Driven Architecture.* PhD thesis, Riga Technical University, RTU Publishing House, Riga, Latvia, 2006.

[5] W.F. Basener. *Topology and Its Applications.* John Wiley and Sons, Inc., New Jersey, USA, 2006. page 339.

[6] A. Cockburn. Structuring use cases with goals. `http://alistair.cockburn.us/crystal/articles/sucwg/`.

[7] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *The Science of Computer Programming*, 20(November):3–50, 1993.

[8] Z. Diskin, B. Kadish, F. Piessens, and M. Johnson. Universal arrow foundations for visual modeling. In *Proc. Diagramms'2000: 1st Int. Conference on the theory and application of diagrams*, pages 345–360. Springer LNAI, 2000. No. 1889.

[9] Eclipse. Eclipse – an open development platform. `http://www.eclipse.org`.

[10] S. Ferg. What's wrong with use cases? `http://www.ferg.org/papers/`, February 2003.

[11] D. Firesmith. Use cases: the pros and cons. `http://www.ksc.com/article7.htm`.

[12] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., Indiana, 2003.

[13] T. Gorschek and C. Wohlin. Requirements abstraction model. *Requirements Engineering*, 11:79–101, 2006.

[14] M. Jackson. The real world. `http://www.ferg.org/papers/`, July 2003.

[15] M. Jackson. Problem frames and software engineering. *Information and Software Technology*, 47(November):903–912, 2005.

[16] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[17] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall PTR, 3rd edition, 2005.

[18] D. Leffingwell and D. Widrig. *Managing Software Requirements: a use case approach.* Addison-Wesley, 2nd edition, 2003.

[19] OMG, `http://www.omg.org/`. *MDA Guide Version 1.0.1*, June 2003.

[20] J. Osis. Extension of software development process for mechatronic and embedded systems. In *Proceeding of the 32nd International Conference*

on Computer and Industrial Engineering, pages 305–310. University of Limerick, Limerick, Ireland, August 2003.

[21] J. Osis. Software development with topological model in the framework of MDA. In *Proceedings of the 9th CaiSE/IFIP8.1/EUNO International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'2004) in connection with the CaiSE'2004*, volume 1, pages 211–220, Riga, Latvia, 2004. Riga Technical University, RTU.

[22] J. Osis. Formal computation independent model within the MDA life cycle. *International Transactions on Systems Science and Applications*, 1(2):159–166, 2006.

[23] H. Podeswa. *UML for the IT Business Analyst: A practical Guide to Object-Oriented Requirements Gathering.* Thomson Course Technology PTR, Boston, 2005.

[24] E.S.K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *International Symposium on Requirements Engineering*, pages 226–235, Annapolis, Maryland, 1997.

e-Informatica Software Engineering Journal, Volume 2, Issue 1, 2008

Integrating Human Judgment and Data Analysis to Identify Factors Influencing Software Development Productivity

Adam Trendowicz*, Michael Ochs*, Axel Wickenkamp*, Jürgen Münch*, Yasushi Ishigai**, Takashi Kawaguchi***

*Fraunhofer Institute for Experimental Software Engineering (Germany)
**Information-Technology Promotion Agency, Software Engineering Center (Japan)
***Toshiba Information Systems Corporation (Japan)

adam.trendowicz@iese.fraunhofer.de, michael.ochs@iese.fraunhofer.de, axel.wickenkamp@iese.fraunhofer.de, juergen.muench@iese.fraunhofer.de, ishigai@ipa.go.jp, kawa@tjsys.co.jp

Abstract

Managing software development productivity and effort are key issues in software organizations. Identifying the most relevant factors influencing project performance is essential for implementing business strategies by selecting and adjusting proper improvement activities. There is, however, a large number of potential influencing factors. This paper proposes a novel approach for identifying the most relevant factors influencing software development productivity. The method elicits relevant factors by integrating data analysis and expert judgment approaches by means of a multi-criteria decision support technique. Empirical evaluation of the method in an industrial context has indicated that it delivers a different set of factors compared to individual data- and expert-based factor selection methods. Moreover, application of the integrated method significantly improves the performance of effort estimation in terms of accuracy and precision. Finally, the study did not replicate the observation of similar investigations regarding improved estimation performance on the factor sets reduced by a data-based selection method.

1. Introduction

Many software organizations are still proposing unrealistic software costs, work within tight schedules, and finish their projects behind schedule and budget, or do not complete them at all [32]. This illustrates that reliable methods for managing software development effort and productivity are a key issue in software organizations.

At the same time, software cost estimation is considered to be more difficult than cost estimation in other industries. This is mainly due to the fact that software organizations typically develop new products as opposed to fabricating the same product over and over again. Moreover, software development is a human-based activity with extreme uncertainties from the outset. This leads to many difficulties in cost estimation, especially during early project phases. To address these difficulties, considerable research has been directed at gaining a better understanding of the software development processes, and at building and evaluating software cost estimation techniques, methods, and tools [8, 34].

One essential aspect when managing development effort and productivity is the large number of associated and unknown influencing factors (so-called *productivity factors*) [33]. Identifying the right productivity factors increases the effectiveness of productivity improvement strategies by concentrating management activities di-

rectly on those development processes that have the greatest impact on productivity. On the other hand, focusing measurement activities on a limited number of the most relevant factors (goal-oriented measurement) reduces the cost of quantitative project management (collecting, analyzing, and maintaining the data). The computational complexity of numerous quantitative methods grows exponentially with the number of input factors [7], which significantly restricts their acceptance in industry.

In practice, two strategies for identifying relevant productivity factors, promoted in the related literature, are widely applied. In *expert-based* approaches, one or more software experts decide about a factor's relevancy [33]. In *data-based* approaches, existing measurement data covering a certain initial set of factors are analyzed in order to identify a subset of factors relevant with respect to a certain criterion [9, 14]. These factor selection strategies have, however, significant practical limitations when applied individually. Experts usually base their decisions on subjective preferences and experiences. In consequence, they tend to disagree by a wide margin and omit relevant factors while selecting irrelevant ones [33]. The effectiveness of data-based methods, on the other hand, largely depends on the quantity and quality of available data. They cannot, for instance, identify a relevant factor if it is not present in the initial set of factors contained by the underlying data set. Moreover, data analysis techniques are usually sensitive to messy (incomplete and inconsistent) data. Yet, assuring that all relevant factors are covered by a sufficient quantity of high-quality measurement data is simply not feasible in practice.

In this paper, we propose an integrated approach to selecting relevant productivity factors for the purpose of software effort estimation. We combine expert- with data-based factor selection methods, using a novel multi-criteria decision aid method called *AvalOn*. The presented approach is then evaluated in the context of a large software organization.

The remainder of the paper is organized as follows. Section 2 provides an overview of factor selection methods. Next, in Section 3, we present the integrated factor selection method, followed by the design of its empirical evaluation (Section 4) and an analysis of the results (Section 5). The paper ends with conclusions (Section 6) and further work perspectives (Section 7).

2. Related Work

2.1. Introduction to Factor Selection

Factor selection can be defined as a process that chooses a subset of M factors from the original space of N factors ($M \leq N$), so that the factor space is optimally reduced according to a certain criterion. In principle, selection process may be based on data analysis, expert' assessments, or both experts and data.

In *expert-based factor selection*, the factor space is practically infinite and not know a priori. There are, in principle, three major types of expert-based factor selection. In the most basic case, experts simply identify a set of most relevant factors without distinguishing the relevance level (factor selection). In addition to selecting the most relevant factors, experts may rank factors (provide order) with respect to their relevancy (factor ranking). Such a ranking does not, however, provide information about the relative distance between certain factors with respect to their relevancy. The most informative way of selecting factors is to quantify their relevancy on the ratio or interval scale. The most common approach is to define factor relevancy on the Likert scale [30] and ask experts to quantify these factors accordingly. In general, quantification could also be done on more than one criterion. In order to select the most relevant factors, quantifications on various criteria have to be aggregated.

In data-based factor selection, the factor space is given a priori and limited by the available measurement data. The data are analyzed in order to identify a limited set of the most relevant factors. Most of the dedicated data-based factor selection methods belong to one of the major areas of the data mining domain, where they are a subclass of the general problem of dimensionality reduction [12] (Figure 1).

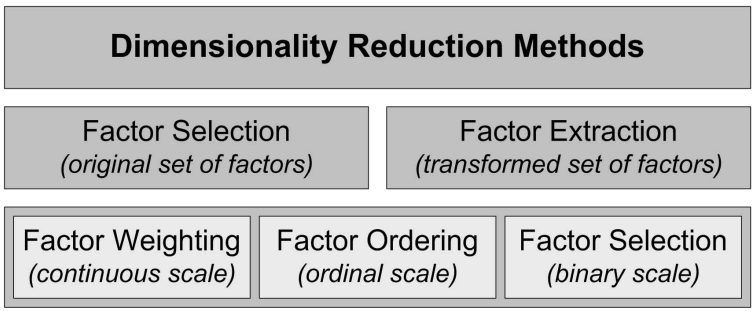


Figure 1. Dimensionality reduction methods

The purpose of dimensionality reduction methods is to reduce a potentially large number of cases and attributes (dimensions) in data in order to limit data noise and the computational time of the analysis. Since the computational complexity of numerous data mining algorithms grows exponentially with the number of dimensions (so-called NP-complete problems), dimensionality reduction allows applying them in practice (they finish within a reasonable amount of time). In this paper, we focus on reducing the number of attributes.

Attribute-oriented dimensionality reduction methods either extract or select factors based on an initial set. *Factor extraction* is a process that extracts a set of M new factors from the original N factors ($M < N$) through some functional mapping (e.g., sum or product). An example factor extraction method is Principal Component Analysis (PCA), which creates new factors as linear transformation of the initial factors. *Factor selection* is a process that extracts a set of M original factors from the original N factors ($M \leq N$).

Moreover, besides a simple subset of factors (factor selection), dimensionality reduction methods may provide an ordered (factor ranking) or weighted (factor weighting) set of factors. *Factor ranking* orders factors with respect to their relevancy, however, no information regarding the distance in rank between subsequent factors is provided (ordinal instead of interval scale).

Factor weighting methods represent the most robust dimensionality reduction approach. They quantify the relative relevance of each i-th factor f_i by providing a ratio-scale weight $w(f_i)$, where usually $w \in [0, 1]$. In this context, factor ranking is weighting on an integer scale (a factor's weight represents its rank) and factor selection is weighting on a dichotomous 0-1 scale (a factor's weight represent its selection or exclusion).

In general, a dimensionality reduction algorithm consists of four basic steps (Figure 2): subset generation, subset evaluation, stopping criterion, and result validation [11].

Subset generation refers to a search (factor selection) or construction (factor extraction) procedure. Basically, it generates subsets of features for evaluation. In case of search procedures, there are various *directions* and *strategies* for searching through the factor space. The most popular search direction, greedy search, comes in three flavors (dependent on the starting point in the search space). Search can start with an empty set and add factors iteratively (*forward selection*) or it can start with a full set (N factors) and be reduced iteratively (*backward elimination*) until the factor subset meets a certain criterion. Hybrid, *bidirectional search* is also possible. Factors can also be searched randomly (*random search*).

The search strategy decides about the scope of the search. *Exhaustive* (*complete*) search performs a complete search through the factor space. Although it guarantees finding the optimal subset of factors, it is usually impractical due to the high computational costs (the problem is known to be NP-hard [2]). *Heuristic search*, as the name suggests, employs heuristics in conducting the search. It avoids being complete, but at the same time risks losing optimal subsets. *Non-deterministic search*, unlike the first two types of strategies, searches for the next set at random (i.e., a current set does not directly

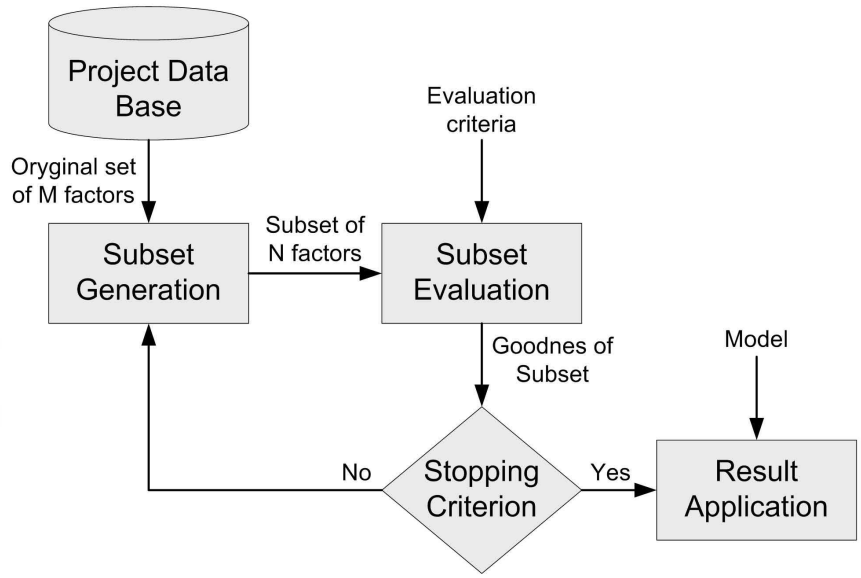


Figure 2. General dimensionality reduction process

grow or shrink from any previous set following a deterministic rule).

The selected subset of factors is always relative to a certain evaluation criterion. Evaluation criteria can be broadly categorized into two groups based on their dependency on the method applied on the selected factors.

In a *filter approach*, the goodness of a factor subset is evaluated according to criteria independent of the method that will later be applied on those factors (i.e., without the involvement of this method). The most common independent criteria are distance measure, information measure, dependency measure, consistency measure, and similarity measure. *Distance measures* are also known as separability, divergence, or discrimination measures. For a given objective factor Z, a factor X is preferred to another factor Y if X induces a greater difference between the conditional probabilities of Z's values than Y; if the difference is zero, then X and Y are indistinguishable. An example is the Euclidean distance measure. Information measures typically determine the information gain related to a certain factor. The information gain from a factor X is defined as the difference between the prior uncertainty and expected posterior uncertainty using X. Factor X is preferred to factor Y if the information gain from factor X is greater than that from factor Y. An example is the entropy measure [23]. *Dependency measures* or correlation measures qualify the ability to predict the value of one variable from the value of another. The coefficient is a classical dependency measure and can be used to find the correlation between a factor and an objective factor Z. If the correlation of factor X with Z is higher than the correlation of factor Y with Z, then factor X is preferred to Y. A slight variation of this is to determine the dependence of a factor on other factors; this value indicates the degree of redundancy of the factor. All evaluation functions based on dependency measures can be theoretically divided into distance and information measures, but they are usually kept as a separate category, because conceptually, they represent a different viewpoint. Finally, *consistency measures* are relatively new and have been in much focus recently. They rely heavily on the training dataset and the use of the Min-Factors bias in selecting a subset of factors. Min-Factors bias prefers consistent hypotheses definable over as few factors as possible. These measures find out the minimally sized subset that satisfies the acceptable inconsistency rate that is usually set by an expert.

In the *wrapper approach*, the goodness of the proposed factor subset is assessed by applying on it and evaluating the performance of the

same method that will by applied on the subset selected eventually. In case of estimation, the well-known Mean Magnitude of Relative Error (MMRE) or the Prediction Level (Pred.25) [10] can be applied.

In an *embedded approach*, factor selection is a part of the learning (model building) process. One classical example is selecting factors along the paths of the Classification and Regression Tree (CART) model [6].

2.2. Factor Selection for Effort Estimation

The purpose of factor selection methods in software effort estimation is to reduce a large number of potential productivity factors (cost drivers) in order to improve estimation performance while maintaining (or reducing) estimation costs. Moreover, information on the most relevant influence factors may be used to guide measurement and improvement initiatives. In practice (authors' observation), relevant cost drivers are usually selected by experts and the selection process is often limited to uncritically adopting factors published in the related literature. Software practitioners adopt the complete effort model along with the integrated factors set (e.g., COCOMO [4]) or build their own model on factors adapted from an existing model. In both situations, they risk collecting a significant amount of potentially irrelevant data and getting limited performance of the resulting model.

Factors uncritically adopted from other contexts most often do not work well leading to much disappointment. They usually contain many irrelevant factors that do not contribute to explaining development productivity variance and increase the cost of data collection, analysis, and maintenance. On the other hand, even though context-specific factors are selected by experts, they tend to disagree largely with respect to the selected factors and their impact on productivity (relevancy).

During the last two decades, several data-based approaches have been proposed to support software organizations that are already collecting data on arbitrarily selected factors in selecting relevant factors. Various data analysis techniques were proposed to simply reduce the factor space by excluding potentially irrelevant factors (factor selection). The original version of the ANGEL tool [27] addressed the problem of optimal factor selection by exhaustive search. However, for larger factor spaces (>15–20), analysis becomes computationally intractable due to its exponential complexity. Alternative, less computationally expensive factor selection methods proposed in the literature include Principal Component Analysis (PCA) [31], Monte Carlo simulation (MC) [16], general linear models (GLM) [18], and wrapper factor selection [14, 9]. The latter approach was investigated using various evaluation models (e.g., regression [9], case-based reasoning [14]), and different search strategies (forward selection [9, 14], as well as random selection and sequential hill climbing [14]). In all studies, a significant reduction (by 50%–75%) of an initial factor set and improved estimation accuracy (by 15%–379%) were reported. Chen et al. [9] conclude, however, that despite substantial improvements in estimation accuracy, removing more than half of the factors might not be wise in practice, because it is not the only decision criterion.

An alternative strategy for removing irrelevant factors would be assigning weights according to a factor's relevancy (*factor weighting*). The advantage of such an approach is that factors are not automatically discarded and software practitioners obtain information on the relative importance of each factor, which they may use to decide about the selection/exclusion of certain factors. Auer et al. [3] propose an optimal weighting method in the context of the k-Nearest Neighbor (k-NN) effort estimator; however, exponential computational complexity limits its practical applicability for large factor spaces. Weighting in higher-dimensionality environments can be, for instance, performed using one of the heuristics based on rough set analysis, proposed recently in [15]. Yet, their application requires additional overhead to discretize continuous variables.

A first attempt towards an integrated factor selection approach was presented in [5],

where Bayesian analysis was used to combine the weights of COCOMO II factors based on human judgment and regression analysis. Yet, both methods were applied on sets of factors previously limited (arbitrarily) by an expert. Moreover, experts weighted factor relevancy on a continuous scale, which proved to be difficult in practice and may lead to unreliable results [35]. Most recently, Trendowicz et al. proposed an informal, integrated approach to selecting relevant productivity factors [35]. They used an analysis of existing project data in an interactive manner to support experts in identifying relevant factors for the purpose of effort estimation. Besides increased estimation performance, the factor selection contributed to increased understanding and improvement of software processes related to development productivity and cost.

3. An Integrated Factor Selection Method

In this paper, we propose an integrated method for selecting relevant productivity factors. The method employs a novel multi-criteria decision aid (MCDA) technique called *AvalOn* to combine the results of data- and expert-based factor selection.

3.1. Expert-based Factor Selection

Expert-based selection of relevant productivity factors is a two-stage process [36]. First, a set of candidate factors is proposed during a group meeting (brainstorming session). Next, factor relevancy criteria are identified and quantified on a Likert scale. Example criteria may include a factor's impact, difficulty, or controllability. *Impact* reflects the strength of a given factor's influence on productivity. *Difficulty* represents the cost of collecting factor-related project data. Finally, *controllability* represents the extent to which a software organization has an impact on the factor's value (e.g., a customer's characteristics are hardly controllable). Experts are then asked to individually evaluate the identified factors according to specified criteria and corresponding measurement scales. In a simple case (when expert involvement has to be limited due to related manpower costs), a simple factor's ranking with respect to its impact on cost (significance) might be performed.

3.2. Data-based Factor Selection

Data-based selection of relevant productivity factors employs one of the available factor weighting techniques. As compared to simple factor selection or ranking techniques, weighting provides experts with the relative distance between subsequent factors regarding their relevance. Selected weighting should be applicable to regression problems, i.e., to the continuous dependent variable (here: development productivity). We recommend excluding factor extraction as well as embedded and wrapper approaches. Factor extraction methods create new set of abstract factors that are not understandable by experts and require additional analysis to gain insight into the relationship between the original factors. There are several arguments for preferring the filter strategy over the wrapper and embedded approaches. Filters (e.g., those based on mutual information criteria) provide a generic selection of variables, not tuned (biased) for/by a given learning machine. Moreover, filtering operates independently of the prediction method, reducing the number of features prior to estimation. Therefore, they can be used as a preprocessing step for reducing space dimensionality and overcoming over-fitting. Finally, filters tend to be far less computationally intensive than wrappers, which run the estimation method in each factor selection cycle.

As far as the search strategy is concerned, although forward selection (FSS) is computationally more efficient than backward selection (BSS), weaker subsets are found by FSS because the importance of variables is not assessed in the context of other variables not included yet [12]. A BSS method may outsmart FSS by eliminating, in the first step, the factor that by itself provides the best performance (explanation of productivity variance, effort estimation accuracy, etc.) in order to retain these two

factors that together perform best. Still, if a very small set of optimal factors (e.g., single best factor) is preferred, or a huge set of initial factors has to be analyzed, BSS would probably be a better alternative [1]. Since software engineering data do not usually cover a large number of factors [35] and usually contains numerous interactions, the BSS strategy should be preferred. One may consider applying the optimal weighting approach as presented in [3]. Due to its significant computational complexity and optimal factor weighting, it might, however, not always be feasible (large factor spaces).

Given the size of the factor space, optimal weighting [3] (small size) or weighting heuristics [15] (large size) should be considered. In this paper, we employ the *Regression ReliefF* (RRF) technique [24]. RRF is well suited for software engineering data due to its robustness against sparse and noisy data. The output of RRF (weighting) reflects the ratio of change to productivity explained by the input factors.

3.3. An Integrated Factor Selection Method

Integrated factor selection combines the results of data- and expert-based selections by means of the AvalOn MCDA method. It is the hierarchically (tree) structured model that was originally used in COTS (Commercial-of-the-shelf) software selection [20, 22, 21]. AvalOn incorporates the benefits of a tree-structured MCDA model such as the Analytic Hierarchy Process (AHP) [25] and leverages the drawbacks of pair-wise (subjective) comparisons. It comprises, at the same time, subjective and objective measurement as well as the incorporation of uncertainty under statistical and simulation aspects. In contrast to the AHP model, which only knows one node type, it distinguishes several node types representing different types of information and offering a variety of possibilities to process data. Furthermore, AvalOn offers a weight rebalancing algorithm mitigating typical hierarchy-based difficulties originating from the respective tree structure. Finally, it allows for any modification (add, delete) of the set of alternatives while maintaining consistency in the preference ranking of the alternatives.

3.3.1. Mathematical background of the AvalOn method

As in many MCDA settings [36], a preference among the alternatives is processed by summing up $weight \cdot preference$ of an alternative. In AvalOn (3), this is accomplished for the node types *root*, *directory*, and *criterion* by deploying the following abstract model in line with the meta-model structure presented in Equation 1.

$$\sum_{j \in \text{subnodes}(i)} w_j \cdot pref_j(a) \qquad (1)$$

where i is a node in the hierarchy, a the alternative under analysis, $subnodes(i)$ the set of child/subnodes of node i, $pref_j(a) \in [0..1]$ the preference of a in subnode j, and $w_j \in [0..1]$ the weight of subnode j. Hence $pref_i(a) \in [0..1]$.

In each model, a value function (*val*) is defined, building the relation between data from {*metrics x alternatives*} and the assigned preference values. *val* may be defined almost in an arbitrary way, i.e., it allows for preference mappings of metric scaled data as well as categorical data.

In this way, *val* can model the whole range of scales from semantic differential via Likert to agreement scales. Please note that when calculating $pref_i(a)$ on the lowest criterion level, the direct outputs of the function *val* in the subnodes, which are *models* in this case, are weighted and aggregated. The full details of the general model definition for *val* is described in [26]. In this context, two examples for *val*, one metric scaled (figure on the left) and one categorical (figure on the right), are given in Figure 4. On the x-axis, there are the input values of the respective metric, while the y-axis shows the individual preference output *val*. A full description of the AvalOn method can be found in [26, 22].

3.3.2. Application of the AvalOn method

AvalOn allows for structuring complex information into groups (element *directory* in

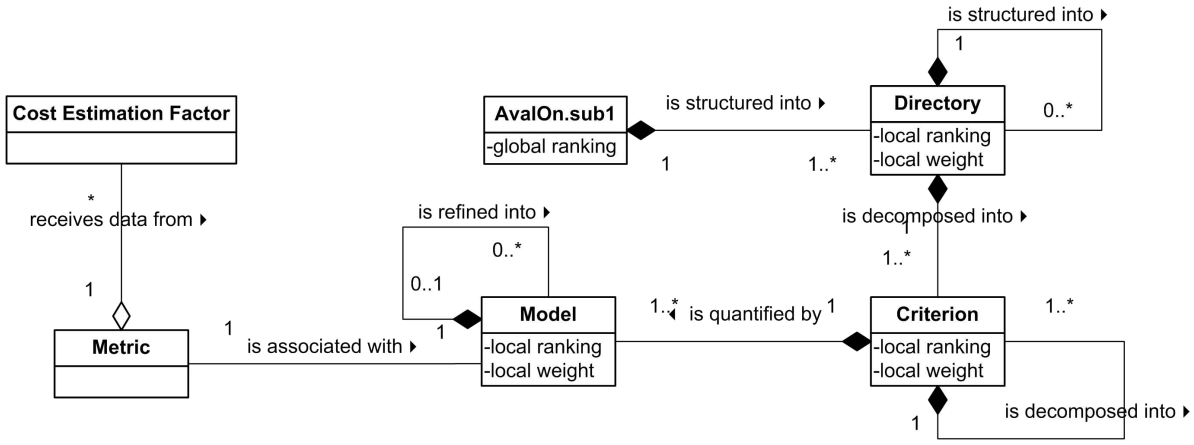


Figure 3. Meta-model for factor selection

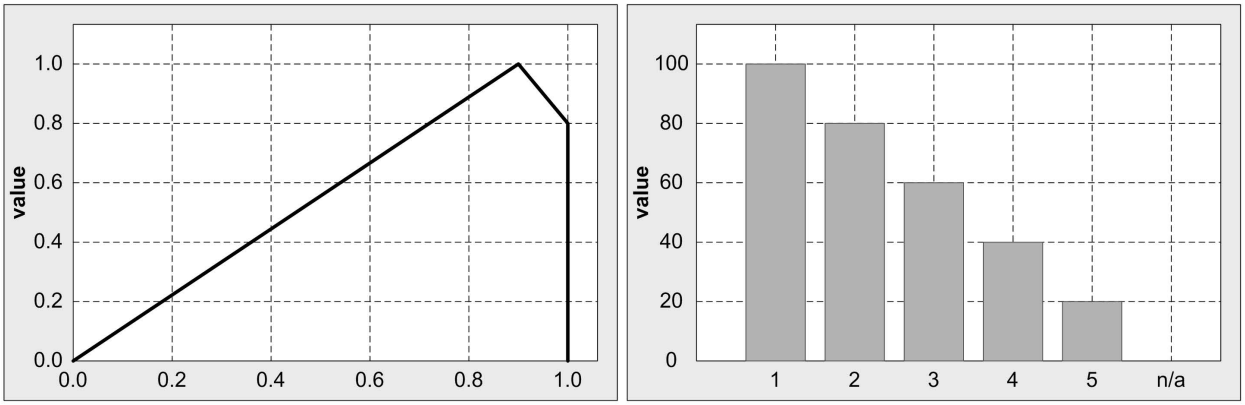


Figure 4. Example *val* models

the meta-model) and criteria (element *criterion* in the meta-model). Each directory as well as each criterion may be refined into sub-directories and sub-criteria. Each (sub)-criterion may then be refined into individual model(s) and sub-models. The models transform the measurement data coming from each alternative into initial preference values. The models providing the preferences based on each measurement by alternative are associated with a set of previously defined metrics. Bottom-up, the data coming from each alternative for potential selection (here: *productivity factors*) are then processed through the models and aggregated from there into the criteria and directory level(s). Finally, in the root node (here: *AvalOn.sub1*), the overall preference of the productivity factors based on their data about individual metrics is aggregated using a weighting scheme that is also spread hierarchically across the tree of decision (selection) criteria. The hybrid character of the setting in this paper can be modeled by combining expert opinion and objective data from, e.g., preliminary data analyses, into criteria and models within different directories, and defining an adequate weighting scheme.

3.3.3. Result Views of the AvalOn Meta-Model

The AvalOn meta-model offers a tool-based variety of result views of the preferences of the alternatives [26]. Major views are available for every node of type *root*, *directory*, *criterion*, and *model* in the criteria hierarchy. In detail views are: (i) overall preference (Figure 5), (ii) node and subnode preference profile (Figure 6), (iii) overall preference range/uncertainty (Figure 7), (iv) prefer-

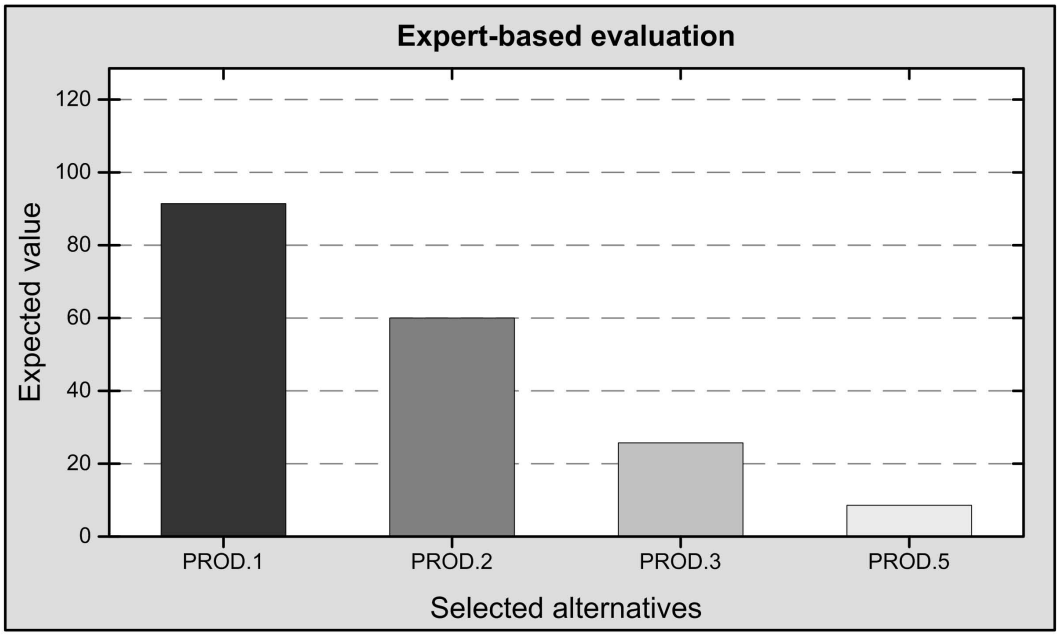


Figure 5. Overall preference

ence weight sensitivity (Figure 8), and (v) pairwise preference comparability (Figure 9).

The *overall preference* view plainly shows the total preference of the alternatives in a selected node of the hierarchy. The *node and subnode preference profile* visualizes the preferences of the alternatives in the selected node on the left line and the individual preferences of the alternatives in the subnodes of the selected node on the lines to the right hand side of it.

The *overall preference range (uncertainty)* view provides graphical information about the variability range of the preferences of the alternatives in a selected node of the hierarchy when evaluation and decision have to be made under uncertainty. In addition, a *t*-test [28] can be performed in order to verify or falsify whether two alternatives do have a significant difference in their individual preferences. The *weight sensitivity* of the *preference* of the alternatives in the selected node analyzes the change of the total preferences of the alternatives when changing the current weight (marked by the vertical line in Figure 8) of one specific subnode. Whenever lines of alternatives intersect at a specific point they are of identical preference, and by continuing in the change direction ofthe subnode weight, the preference between the alternatives will be changed (for two alternatives: turned around).

The pairwise preference comparability view shows for each pair of alternatives – in a direct comparison – whether one of the alternatives is to be preferred over the other (green areas), whether the two alternatives are indifferent (white area), or whether they are incomparable (yellow area). This method deploys ORESTE+ [26], which is a metric relaxation of the ordinal ORESTE procedure.

4. Empirical Study

The integrated factor selection method proposed in this paper was evaluated in an industrial context. We applied the method for the purpose of software effort estimation and compared it with isolated expert- and data-based selection methods. Data-based factor selection employed the RRF technique [24] implemented in the WEKA data mining software [37]. Expert-based factor selection was performed as a multiple-expert ranking (see Section 3.1 regarding the ranking process).

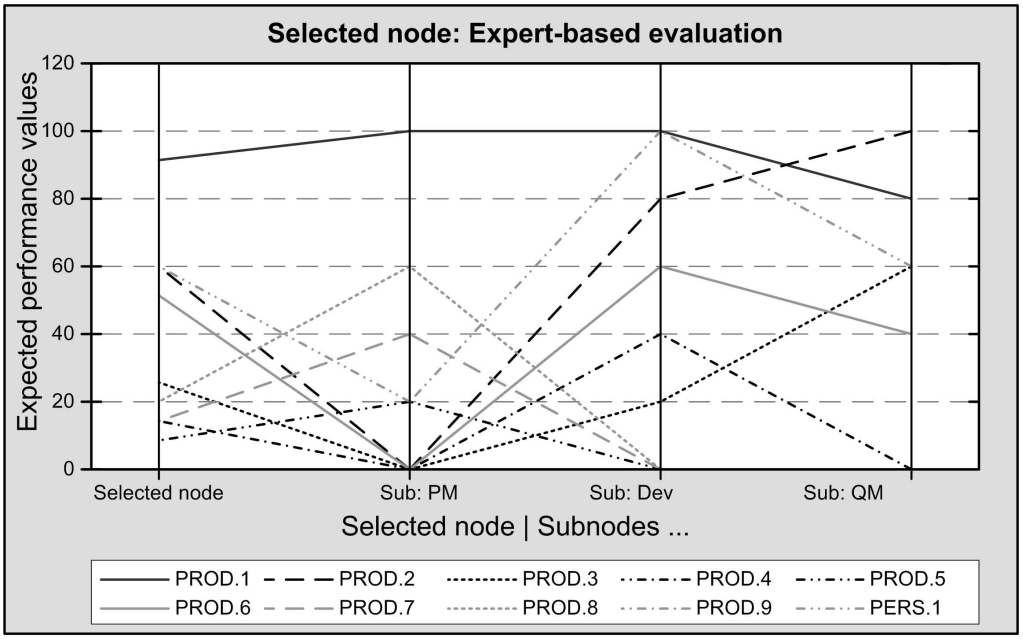


Figure 6. Node and subnode preference profile

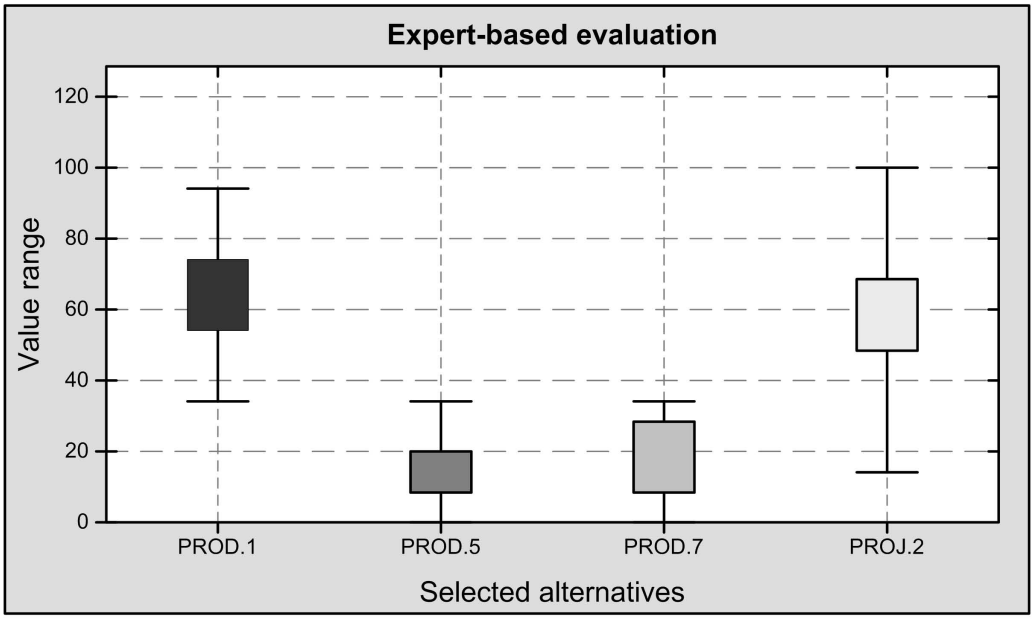


Figure 7. Overall preference range/uncertainty

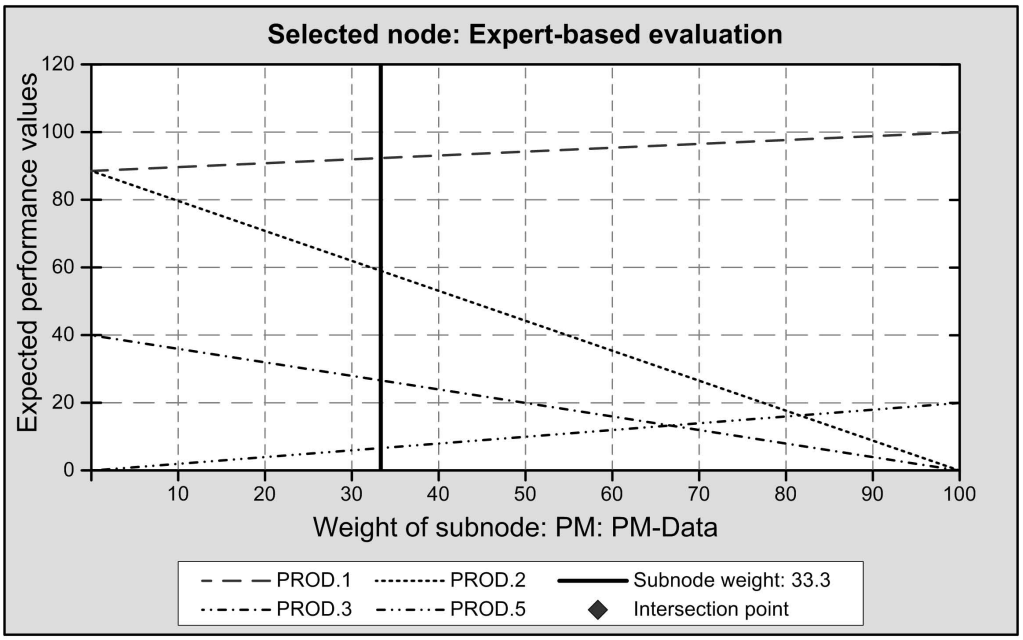


Figure 8. Preference weight-sensitivity

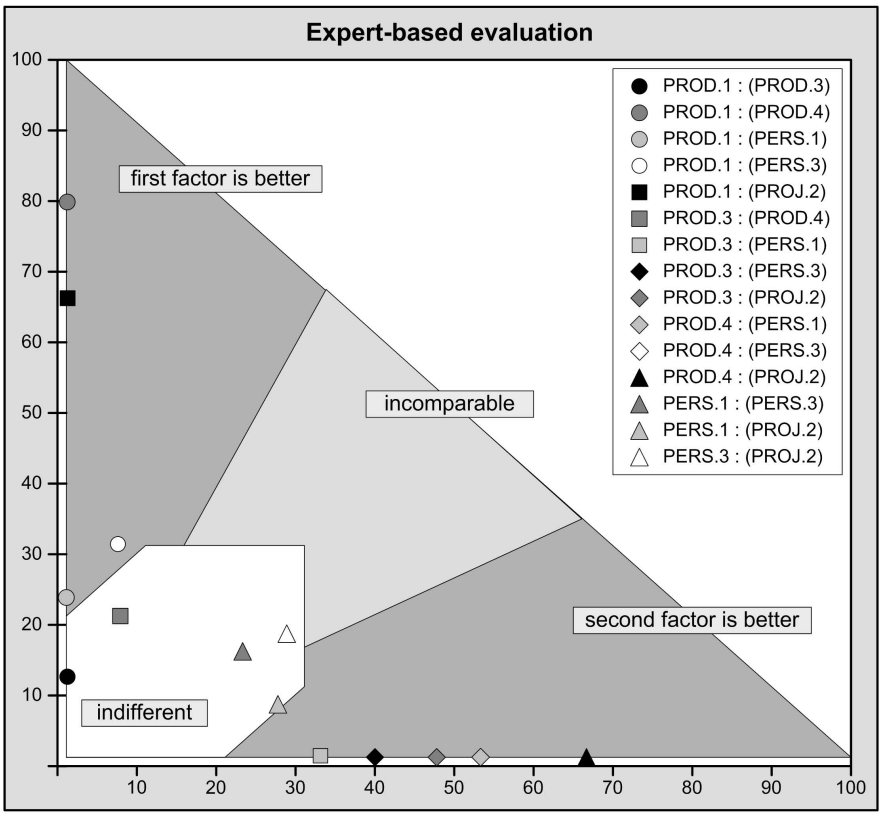


Figure 9. Pairwise preference comparability

4.1. Study Objectives and Hypotheses

The objective was to evaluate, in a comparative study, expert- and data-based approaches and the integrated approach for selecting the most relevant productivity factors in the context of software effort estimation. For that purpose, we defined two research questions and related hypotheses:

Q1. *Do different selection methods provide different sets of productivity factors?*

H1. Expert-based, data-based, and integrated methods select different (probably partially overlapping) sets of factors.

Q2. *Which method (including not reducing factors at all) provides the better set of factors for the purpose of effort estimation?*

H2. The integrated approach provides a set of factors that ensure higher performance of effort estimation than factors provided by expert- and data-based selection approaches when applied individually.

Some effort estimation methods such as stepwise regression [10] or OSR [9] already include embedded mechanisms for selecting relevant productivity factors. In our study, we wanted to evaluate in addition how preliminary factor selection done by an independent method influences the performance of such estimation methods. This leads us to a general research question:

Q3. *Does application of an independent factor selection method increase the prediction performance of an estimation method that already has an embedded factor selection mechanism?*

Answering such a generic question would require evaluating all possible estimation methods. This, however, is beyond the scope of this study. We limit our investigation to the OSR estimation method [9] and define a corresponding research hypothesis:

H3. Application of an independent factor selection method does not increase the prediction performance of the OSR method.

Finally, in order to validate and replicate the results of the most recent research regarding the application of data-based factor selection to analogy-based effort estimation (e.g., [9, 14]), we define the following question:

Q4. *Does application of a data-based factor selection method increase the prediction performance of an analogy estimation method?*

H4. Application of a data-based factor selection method increases the prediction performance of a k-NN estimation method.

4.2. Study Context and Empirical Data

The empirical evaluation was performed in the context of Toshiba Information Systems (Japan) Corporation (TJSYS). The project measurement data repository contained a total of 76 projects from the information systems domain. Figure 10 illustrates the variance of development productivity measured as function points (unadjusted, IFPUG) per man-month.

Expert assessments regarding the most relevant factors were obtained from three experts (see Table 1). During the group meeting (brainstorming session), an initial set of factors was identified. It was then grouped into project-, process, personnel-, and product-related factors as well as context factors. The first four groups refer to the characteristics of the respective entities (software project, development process, products, and stakeholders). The latter group covers factors commonly used to limit the context of software effort estimation or productivity modeling. The *application domain*, for instance, is often regarded as a context factor, i.e., an effort model is built for a specific application domain. Finally, experts were asked to select the 5 most important factors from each category and rank them from most relevant (rank = 1) to least relevant (rank = 5).

4.3. Study Limitation

Unfortunately, the measurement repository available did not cover all relevant factors selected by the experts. It was also not possible to collect the data ex post facto. This prevented us from doing a full comparative evaluation of the three factor selection methods considered here for the purpose of software effort estimation. In order to at least get an indication of the methods' performance, we decided to compare them (instead

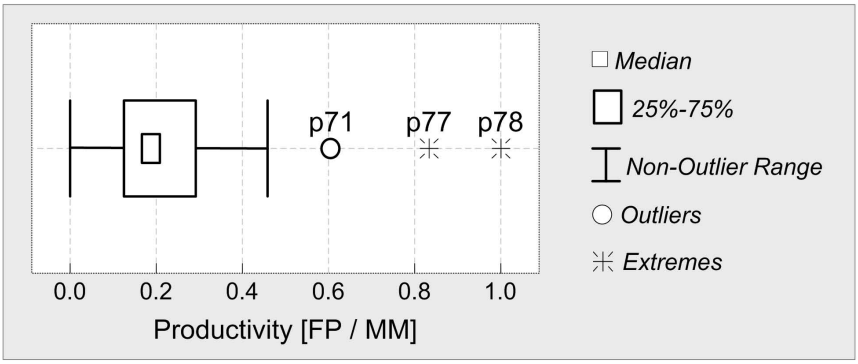


Figure 10. Development productivity variance; data presented in a normalized form

Table 1. Experts who participated in the study

	Expert 1	**Expert 2**	**Expert 3**
Position/Role	Project manager	Developer	Quality manager
Experience [#working years]	8	15	3
Experience [#performed projects]	30	15	40

of all identified factors) on the factors identified by experts for which measurement data were available. This would represent the situation where those factors cover all factors available in the repository and identified by experts.

4.4. Study Design and Execution

4.4.1. Data Preprocessing

Measurement data available in the study suffered from incompleteness (44.3% missing data). An initial preprocessing was thus required in order to apply the data analysis techniques selected in the study. We wanted to avoid using simple approaches to handling missing data such as list-wise deletion or mean imputation, which significantly reduce data quantity and increase noise. Therefore, we decided to apply the *k-Nearest Neighbor* (k-NN) imputation method. It is a common *hot deck* method, in which k nearest projects minimizing a certain similarity measure (calculated on non-missing factors) are selected to impute missing data. It also proved to provide relatively good results when applied to sparse data in the context of software effort prediction [19]. Moreover, other more sophisticated (and potentially more effective) imputation methods required removing factor collinearities beforehand. Such a preprocessing step would, however, already be a kind of factor selection and might thus bias the results of the actual factor selection experiment. We adopted the k-NN imputation approach presented in [13].

We assumed a *missing at random* (*MAR*) missingness mechanism, which means [17] that the cause of the missing data is completely unrelated to the missing values; it may be related to the observed values of other variables. This assumption is weaker than *missing completely at random* (*MCAR*); however, it is more realistic and seems not to have a significant impact on the accuracy of the k-NN imputation method [29].

In order to assure maximal performance of the imputation, before applying it, we removed factors and projects with a large missing data ratio so that the total ratio of missing data was reduced to around one third; however, with minimal loss of non-missing data. We applied the following procedure: We first removed factors where 90% of the data were missing and next, projects where more than 55% of the data were still missing. As a result, we reduced the total rate of missing data to 28.8%, while losing a minimal quantity of information (removed 19 out of 82 factors and 3 out of 78 projects). The remaining 28.8%

of missing data were imputed using the k-NN imputation technique.

4.4.2. Empirical Evaluation

Let us first define the following abbreviations for the factor sets used in the study:

FM: factors covered by measurement data.

$\mathbf{FM_R}$: relevant FM factors selected by the RReliefF method (factors with $weight > 0$).

$\mathbf{FM_{R10}}$: the 10% most relevant FMR factors.

FE: factors selected by experts.

FI: factors selected by the integrated method.

FT: all identified factors ($FM \cup FE$).

FC: factors selected by experts for which measurement data are available ($FM \cap FE$).

$\mathbf{FC_{E25}}$: the 25% most relevant FC factors selected by experts.

$\mathbf{FC_{R25}}$: the 25% most relevant FC factors selected by the RRF method.

$\mathbf{FC_{I25}}$: the 25% most relevant FC factors selected by the integrated method.

Hypothesis H1. In order to evaluate H1, we compared factor sets selected by the data-based, expert-based, and integrated methods ($\mathrm{FM_R}$, FE, and FI). For the 10 most relevant factors shared by all three factor sets, we compared the ranking agreement using Kendall's coefficient of concordance [28].

Hypothesis H2. In order to evaluate H2, we evaluated the estimation performance of two data-based estimation methods: *k-Nearest Neighbor* (k-NN) [27] and *Optimized Set Reduction* (*OSR*) [7]. We applied them in a leave-one-out cross validation on the following factor sets: FM, FC, $\mathrm{FC_{E25}}$, $\mathrm{FC_{R25}}$, and $\mathrm{FC_{I25}}$.

Hypothesis H3. In order to evaluate H3, we compared the estimation performance of OSR (which includes an embedded, data-based factor selection mechanism) when applied on the FM and $\mathrm{FM_{R10}}$ factor sets.

Hypothesis H4. In order to evaluate H4, we compared the estimation performance of the k-NN method when applied on the FM and $\mathrm{FM_{R10}}$ factor sets.

To quantify the estimation performance in H2, H3, and H4, we applied the common accuracy and precision measures defined in [10]: *magnitude of relative estimation error* (*MRE*), *mean* and *median of MRE* (*MMRE* and *MdMRE*), as well as *prediction at level 25%* (*Pred.25*). We also performed an *analysis of variance* (*ANOVA*) [28] of MRE to see if the error for one approach was statistically different from that of another one. We interpret the results as being statistically significant if the results could be due to chance less than 2% of the time ($p < 0.02$).

5. Results of the Empirical Study

Hypothesis H1*: Expert-based, data-based and integrated methods select different (probably partially overlapping) sets of factors.*

After excluding the dependent variable (development productivity) and project ID, the measurement repository contained data on 61 factors. Experts identified a total of 34 relevant factors, with only 18 of them being already measured (FC). The RRF method selected 40 factors ($\mathrm{FM_R}$), 14 of which were also selected by experts. The integrated approach selected 59 factors in total, with only 14 being shared with the former two selection methods. Among the FC factors, as many as 8 were ranked by each method within the top 10 factors (Table 2). Among the top 25% FC factors selected by each method, only one factor was in common, namely *customer commitment and participation*. There was no significant agreement (Kendall = 0.65 at $p = 0.185$) between data- and expert-based rankings on the FC factors. The integrated method introduced significant agreement on ranks produced by all three methods (Kendall = 0.72 at $p = 0.004$).

Interpretation (H1): Data- and expert-based selection methods provided different (partially overlapping) sets of relevant factors. Subjective evaluation of the shared factors suggests that both methods vary regarding the assigned factor's importance; yet this could not be confirmed by statistically significant results. The

Table 2. Comparison of the ranks on FC factors (top 25% marked in bold)

Productivity factor	$\mathbf{FC_E}$	$\mathbf{FC_R}$	$\mathbf{FC_I}$
Customer commitment and participation	**3**	**3**	**3**
System configuration (e.g., client-server)	5	**2**	5
Application domain (e.g., telecommunication)	**1**	6	**1**
Development type (e.g., enhancement)	7	**1**	**4**
Application type (e.g., embedded)	**2**	7	**2**
Level of reuse	9	**4**	9
Required product quality	6	10	7
Peak team size	8	9	8

integrated method introduced a consensus between individual selections (significant agreement) and as such might be considered as a way to combine the knowledge gathered in experts' heads and in measurement data repositories.

Hypothesis H2*: The integrated approach provides a set of factors that ensure higher performance of effort estimation than factors provided by expert- and data-based selection approaches when applied individually.*

A subjective analysis of the estimates in Table 3 suggests that the k-NN provided improved estimates when applied on a reduced FC factors set (FC_{E25}, FC_{R25}, and FC_{I25}), whereas OSR does not consistently benefit from independent factor reduction (by improved estimates). The analysis of the MRE variance, however, showed that the only significant ($p = 0.016$) improvement in estimation performance of the k-NN predictor was caused by the integrated factor selection method. The OSR predictor improved its estimates significantly ($p < 0.02$) only on the FC_{E25} factors set.

Interpretation (H2): The results obtained indicate that a factor set reduced through an integrated selection contributes to improved effort estimates. Yet, this does not seem to depend on any specific way of integration. The k-NN predictor, which uses all input factors, improved on factors reduced by the AvalOn method. The OSR method, however, improved slightly on the factors reduced by experts. This interesting observation might be explained by the fact that OSR, which includes an embedded, data-based factor selection mechanism, combined this with prior expert-based factor selection. Still, the effectiveness of such an approach largely depends on the experts who determine (pre-select) input factors for OSR (expert-based selection is practically always granted higher priority).

Hypothesis H3*: Application of an independent factor selection method does not increase the prediction performance of the OSR method.*

A subjective analysis of OSR's estimation error (Table 3 and Table 4) suggests that it performs generally worse when applied on the factors chosen by an independent selection method. This observation was, however, not supported by the analysis of the MRE variance. The exception was the FC set reduced by experts (FC_{E25}), on which a slight, statistically significant improvement of the OSR's predictions was observed.

Interpretation (H3). The results obtained indicate that no general conclusion regarding the impact of independent factor selection on the prediction performance of OSR can be drawn. Since no significant deterioration of estimation performance was observed, application of OSR on the reduced set of factors can be considered useful due to the reduced cost of measurement. Yet, improving OSR's estimates might require a selection method that is more effective than the selection mechanism embedded in OSR.

Hypothesis H4*: Application of a data-based factor selection method increases the prediction performance of a k-NN estimation method.*

A subjective impression of improved estimates provided by the k-NN predictor (Table 4) when applied on the reduced factor set (FM_{R10}) was, however, not significant in the sense of different variances of MRE ($p = 0.39$). Yet, estimates provided by the k-NN predictor improved significantly when used on the FC data set reduced by the integrated selection method ($p = 0.016$). The two individual selection methods did not

Table 3. Comparison of various factor selection methods

Predictor	Factors Set	MMRE	MdMRE	Pred.25
k-NN	FM	73.7%	43.8%	21.3%
	FC	52.6%	40.0%	26.7%
	FC_{E25}	46.3%	38.5%	33.3%
	FC_{R25}	48.3%	36.9%	29.3%
	FC_{I25}	**47.5%**	**33.3%**	**30.7%**
OSR	FM	59.7%	50.8%	17.3%
	FC	65.9%	59.2%	18.7%
	FC_{E25}	**30.7%**	**57.9%**	**24.0%**
	FC_{R25}	66.2%	52.1%	14.7%
	FC_{I25}	65.1%	57.9%	14.7%

Table 4. Results of data-based factor selection

Predictor	Factors Set	MMRE	MdMRE	Pred.25	ANOVA
k-NN	FM	73.7%	43.8%	21.3%	$p = 0.39$
	FM_{R10}	56.8%	40.7%	22.7%	
OSR	FM	59.7%	50.8%	17.3%	$p = 0.90$
	FM_{R10}	68.1%	59.1%	16.0%	

significantly improve performance of the k-NN predictor.

Interpretation (H4): Although a subjective analysis of the results (Table 3 and Table 4) suggests improved estimates provided by the k-NN predictor when applied on reduced factors sets, no unambiguous conclusion can be drawn. The performance of k-NN improved significantly only when applied on factors identified from the FC set by the integrated selection method (the FC_{I25} set). This might indicate that k-NN's performance improvement depends on the applied factor selection method (here, the integrated method was the best one).

Threats to Validity. We have identified two major threats to validity that may limit the generalizability of the study results. First, the estimation performance results of the factor selection methods investigated, compared on the FC set, might not reflect their true characteristics, i.e., as compared on the complete set of identified factors (threat to hypothesis H2). Yet, a lack of measurement data prevented us from checking on this. Second, the RRF method includes the k-NN strategy to search through the factor space and iteratively modify factor weights. This might bias the results of k-NN-based estimation by contributing to better performance of k-NN (as compared to OSR) on factors selected by RRF (threat to hypotheses H3 and H4).

6. Summary

In this paper, we proposed an integrated approach for selecting relevant factrs influencing software development productivity. We compared the approach in an empirical study against selected expert- and data-based factor selection approaches.

The investigation performed showed that expert- and data-based selection methods identified different (only partially overlapping) sets of relevant factors. The study indicated that the *AvalOn* method finds a consensus between factors identified by individual selection methods. It combines not only the sets of relevant factors, but also the individual relevancy levels of selected factors. We showed that in contrast to data- and expert-based factor selection methods, the integrated approach may significantly improve the estimation performance of estimation methods that do not include an embedded factor selection mechanism. Estimation methods that include such a mechanism may, however, benefit from integrating their capabilities with expert-based factor selection.

The study did not replicate the observation of similar investigations regarding improved estimation performance on the factor sets reduced by a data-based selection method. Neither of the estimation methods employed in the study (k-NN and OSR) improved significantly when applied on factor sets reduced by the *RReliefF* method. Although k-NN improved in terms of aggregated error measures (e.g., MMRE) the difference in the MRE variance was insignificant. The results obtained for the OSR method may indicate that the change of its prediction performance when applied on a reduced set of factors depends on the selection method used.

Finally, we also observed that the function point adjustment factor (FPAF) was not considered among the most relevant factors, although factor selection was driven by a variance on development productivity calculated from unadjusted function point size. Moreover, some of the factors considered as relevant (e.g., performance requirements) belong to components of the FPAF. This might indicate that less relevant sub-factors of the FPAF and/or the adjustment procedure itself may hide the impact of relevant factors. Considering sub-factors of FPAF individually might therefore be more beneficial.

In conclusion, factor selection should be considered as an important aspect of software development management. Since individual selection strategies seem to provide inconsistent results, integrated approaches should be investigated to support software practitioners in limiting the cost of management (data collection and analysis) and increasing the benefits (understanding and improvement of development processes).

7. Further Work

Further work shall focus on several aspects. First, a full evaluation of the three selection strategies presented on a complete data set (including data on all factors identified by experts) shall be performed.

Daily industrial practice requires an incremental approach to identify relevant productivity factors. After identifying a single most relevant factor or small group of (probably related) most relevant factors, corresponding project data should be collected in order to quantitatively validate the true impact on productivity. The identified factors may, for instance, be applied within an estimation model (such as CoBRA [35]) in order to see how much productivity variance they are able to explain across the considered development projects. This shall be the next subject of our further investigation.

Finally, methods for identifying and explicitly considering factor dependencies need to be investigated. Such information might not only improve performance in effort estimation and productivity modeling, but they also improve understanding of the interaction between organizational processes influencing development productivity.

Acknowledgments. We would like to thank Toshiba Information Systems (Japan) Corporation and all involved experts who greatly contributed to the study. We would also like to thank the Japanese Information-technology Promotion Agency for supporting the study. Finally, we would like to thank Sonnhild Namingha and Marcus Ciolkowski for reviewing the paper.

References

[1] D.W. Aha and R.L. Bankert. A comparative evaluation of sequential feature selection algorithms. In Doug Fisher and Hans-J. Lenz, editors, *Artificial Intelligence and Statistics*, Chapter 4, pages 199–206. Springer-Verlag, 1996.

[2] E. Amaldi and V. Kann. On the approximation of minimizing non zero variables or unsatisfied relations in linear systems. *Theoretical Computer Science*, Volume 209(1-2):237–260, 1998.

[3] M. Auer, A. Trendowicz, B. Graser, E.J. Haunschmid, and S. Biffl. Optimal project feature weights in analogy-based cost estimation: Improvement and limitations. *IEEE Transactions on Software Engineering*, 32(2):83–92, February 2006.

[4] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with CoCoMo II*. Prentice-Hall, 2000.

[6] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Advanced Books & Software. Chapman and Hall, New-York, 1984.

[7] L.C. Briand, V.R. Basili, and W. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(1):931–942, November 1992.

[8] L.C. Briand and I. Wieczorek. Software resource estimation. In J.J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 2, pages 1160–1196. John Wiley & Sons, 2002.

[9] Z. Chen, T. Menzies, D. Port, and B. Boehm. Finding the right data for software cost modeling. *IEEE Software*, 22(6):38–46, November/December 2005.

[10] S. Conte, H. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models*. CA: Benjamin Cummings, 1986.

[11] M. Dash and H. Liu. Feature selection methods for classifications. *An International Journal on Intelligent Data Analysis*, 1(3):131–156, 1997.

[12] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[13] P. Jönsson and C. Wohlin. An evaluation of k-nearest neighbour imputation using likert data. In *Proceedings of the 10th IEEE International Software Metrics Symposium*, pages 108–118. IEEE Computer Society, 2004.

[14] C. Kirsopp, M. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374, 2002.

[15] J. Li and G. Ruhe. A comparative study of attribute weighting heuristics for effort estimation by analogy. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 66–74, 2006.

[16] T. Liang and A. Noore. Multistage software estimation. In *Proceedings of the 35th Southeastern Symposium on System Theory*, pages 232–236, 2003.

[17] R.J.A. Little and D.B. Rubin. *Statistical Analysis with Missing Data*. John Wiley & Sons, New York, 2nd edition, 2002.

[18] K.D. Maxwell, L. Van Wassenhove, and S. Dutta. Software development productivity of european space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, October 1996.

[19] I. Myrtveit, E. Stensrud, and U.H. Olsson. Analyzing data sets with missing data: An empirical evaluation of imputation methods and likelihood-based methods. *IEEE Transactions on Software Engineering*, 27(11):999–1013, November 2001.

[20] M. Ochs. Using sw risk management for deriving method requirements for risk mitigation in cots assessment & selection. In *Proceeding of the International Conference on Software Engineering and Knowledge Engineering*, 2003.

[21] M. Ochs, D. Pfahl, G. Chrobok-Diening, and B. Nothhelfer-Kolb. A cots acquisition process: definition and application experience. In *Proceedings of the 11th European Software Control and Metrics Conference*, 2000.

[22] M. Ochs, D. Pfahl, G. Chrobok-Diening, and B. Nothhelfer-Kolb. A method for efficient measurement-based cots assessment & selection – method description and evaluation results. In *Proceedings of the 7th International Syposium on Software Metrics*, 2001.

[23] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[24] M. Robnik-Sikonja and I. Kononenko. Theoretical and empirical analysis of ReliefF and RReliefF. *The Machine Learning Journal*, 53:23–69, 2003.

[25] T.L. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, New York, 1990.

[26] D. Schillinger. *Entwicklung eines simulationsfähigen COTS Assessment und Selection Tools auf Basis eines für Software adequaten hierarchischen MCDM Meta Modells*. PhD thesis, Deptartment of Computer Science, TU Kaiserslautern, Kaiserslautern, Germany, 2006. Supervisors: Prof. Dr. D.H. Rombach, M. Ochs.

[27] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(12):736–743, November 1997.

[28] D.J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2nd edition, 2000.

[29] Q. Song and M. Shepperd. A short note on safest default missingness mechanism assumptions. *Empirical Software Engineering*, 10(2), 2005.

[30] P. Spector. *Summated Rating Scale Construction*. Sage Publications, 1992.

[31] G.H. Subramanian and S. Breslawski. Dimensionality reduction in software development effort estimation. *Journal of Systems and Software*, 21(2):187–196, 1993.

[32] The Standish Group. CHAOS chronicles. Technical report, The Standish Group, West Yarmouth, MA, 2003.

[33] A. Trendowicz. Factors influencing software development productivity – state of the art and industrial experiences. Technical Report 08.07/E, Fraunhofer IESE, Kaiserslautern, Germany, 2007.

[34] A. Trendowicz. Software effort estimation – overview of current industrial practices and existing methods. Technical Report 06.08/E, Fraunhofer IESE, Kaiserslautern, Germany, 2008.

[35] A. Trendowicz, J. Heidrich, J. Münch, Y. Ishigai, K. Yokoyama, and N. Kikuchi. Development of a hybrid cost estimation model in an iterative manner. In *Proceedings of the 28th International Conference on Software Engineering*, pages 331–340, 2006.

[36] P. Vincke. *Multicriteria Decision-aid.* John Wiley & Sons, Chichester, 1992.

[37] I.H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, 2005.

e-Informatica Software Engineering Journal, Volume 2, Issue 1, 2008

A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams

Philip Samuel*, Rajib Mall*

*Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur(WB), India-721302

philips@cusat.ac.in, rajib@cse.iitkgp.ernet.in

Abstract

We present a novel methodology for test case generation based on UML sequence diagrams. We create message dependence graphs (MDG) from UML sequence diagrams. Edge marking dynamic slicing method is applied on MDG to create slices. Based on the slice created with respect to each predicate on the sequence diagram, we generate test data. We formulate a test adequacy criterion named slice coverage criterion. Test cases that we generate achieves slice coverage criterion. Our approach achieves slice test coverage with few test cases. We generate effective test cases for cluster level testing.

1. Introduction

Ever since Weiser [51] introduced program slicing, researchers have shown considerable interest in this field probably due to its application potential. Slicing is useful in software maintenance and reengineering [14, 33], testing [18, 26, 40], decomposition and integration [23], decompilation [10], program comprehension [36, 19], and debugging [37]. Most of the works reported on slicing concerns improvements and extensions to algorithms for slice construction [35, 20, 31, 50, 6]. Even though dynamic slicing is identified as a powerful tool for software testing [31, 40], reported work on how dynamic slicing can be used in testing is rare in the literature. In 1993, Kamkar et al. [26] reported how dynamic slicing can be applied to interprocedural testing. This work is reported in the context of testing procedural code. To the best of our knowledge, no work is reported in the literature that describes how dynamic slicing can be used for test case generation in the object oriented context. In this paper, we propose a method to generate test cases by applying dynamic slicing on UML sequence diagrams.

As originally introduced, slicing (static slicing) considers all possible executions of a program. Korel and Laski [30] introduced the concept of dynamic slicing. Dynamic slicing considers a particular execution and hence significantly reduces the size of the computed slice. A dynamic slice can be thought of as that part of a program that "affects" the computation of a variable of interest during a program execution on a specific program input [31]. A dynamic slice is usually smaller than a static slice, because run-time information collected during execution is used to compute the slice. In a later work, Korel has shown that slicing can be used as a reduction technique on specifications like UML state models [32].

The goal of software testing is to ensure quality. Software testing is necessary to produce highly reliable systems, since static verification techniques suffer from several handicaps in detecting all software faults [5]. Hence, testing will be a complementary approach to static verification techniques to ensure software quality. As software becomes more pervasive and is used more often to perform critical tasks, it will be

required to be of very high quality. Unless more efficient ways to perform effective testing are found, the fraction of development costs devoted to testing will increaseto unacceptable levels [44].

The most intellectually challenging part of testing is the design of test cases. Test cases are usually generated based on program source code. An alternative approach is to generate test cases from specifications developed using formalisms such as UML models. In this approach, test cases are developed during analysis or design stage itself, preferably during the low level design stage. Design specifications are an intermediate artifact between requirement specification and final code. They preserve the essential information from the requirement, and are the basis of code implementation. Moreover, in component-based software development, often only the specifications are available and the source code is proprietary. Test case generation from design specifications has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. It is therefore desirable to generate test cases from the software design or analysis documents, in addition to test case design using the code.

Now, UML is widely used for object oriented modeling and design. Recently, several methods have been proposed to execute UML models [47, 39, 46, 17, 13, 11]. Executable UML [39, 46] allows model specifications to be efficiently translated into code. Executable UML formalizes requirements and use cases into a set of verifiable diagrams. The models are executable and testable and can be translated directly into code by executable UML model compilers. Besides reducing the effort in the coding stage, it also ensures platform independence and avoids obsolescence. This is so because the code often needs to change when ported to new platforms or fine tuning the code on efficiency or reliability considerations. It also allows meaningful verification of the models by executing them in a test and debug environment. Our test generation approach can also work on executable UML models.

UML-based automatic test case generation is a practically important and theoretically challenging topic. Literature survey indicates, testing based on UML specifications is receiving an increasing attention from researchers in the recent years. In using UML in the software testing process, here we focus primarily on the sequence diagrams where sequence diagrams model dynamic behavior. This is because most of the activities in software testing seek to discover defects that arise during the execution of a software system, and these defects are generally dynamic (behavioral) in nature [52]. Software testing is fundamentally concerned with behavior (what it does), and not structure (what it is) [25]. Customers understand software in terms of its behavior, not its structure. Further, UML is used in the design of object-oriented software, which is primarily event-driven in nature. In such cases, the concept of a main program is minimized and there is no clearly defined integration structure. Thus there is no decomposition tree to impose the question of integration testing order of objects. Hence, it is no longer natural to focus on structural testing orders. Whereas, it is important to identify in what sequence objects interact to achieve a common behavior. In this context, UML sequence diagrams forms an useful means by which we can generate effective test cases for cluster level testing.

In this paper, we concentrate on UML sequence diagrams to automatically generate test cases. This paper is organized as follows: A brief discussion on sequence diagrams is given in the next section. In Section 3 we discuss few basic concepts. Section 4 describes our methodology to generate test cases from sequence diagrams and explains our methodology with an example. Section 5 discusses an implementation of our test methodology. Related research in the area of UML based testing is discussed in the Section 6 and conclusions are given in Section 7.

2. UML Sequence Diagrams

UML Sequence diagrams capture time dependent (temporal) sequences of interactions between objects. They show the chronological sequence of the messages, their names and responses and their possible arguments. A sequence diagram

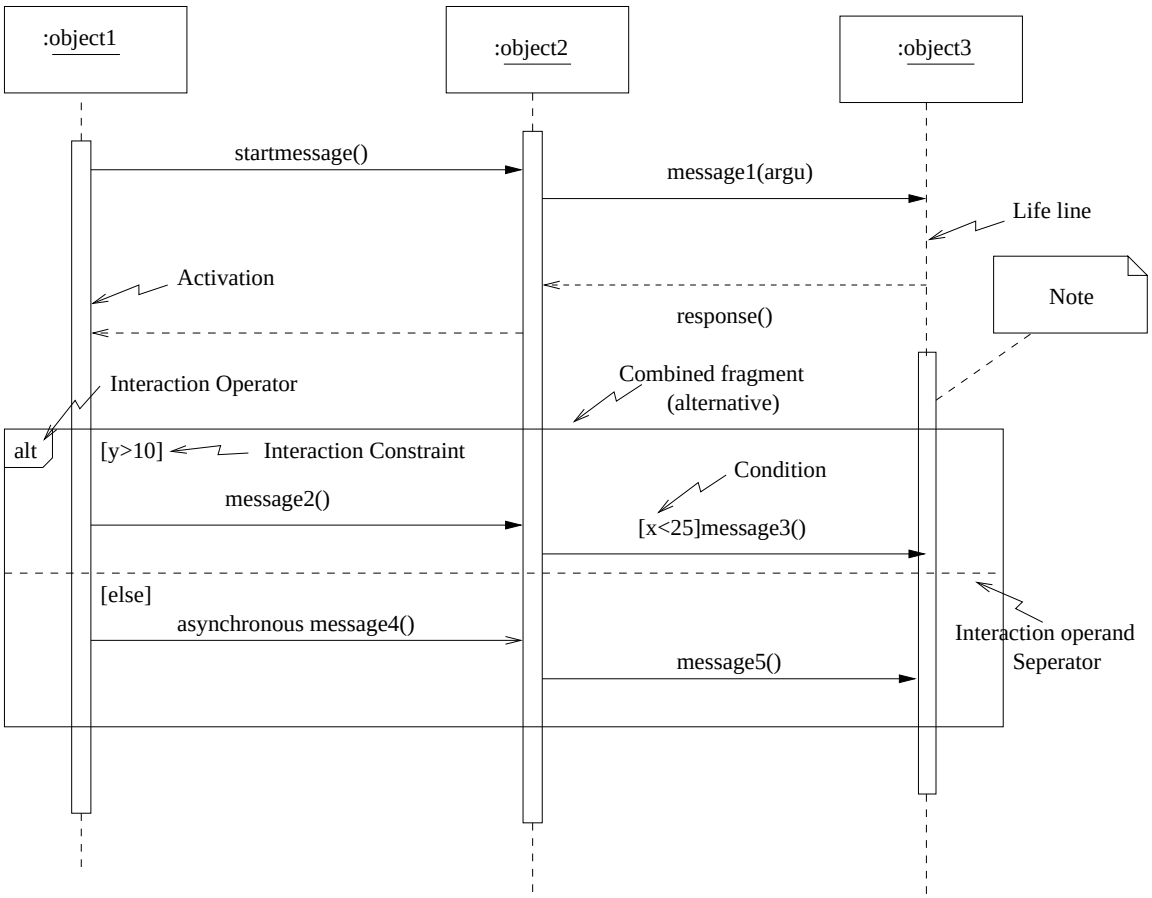


Figure 1. A sequence diagram showing various notations

has two dimensions: the vertical dimension represents time, and the horizontal dimension represents different instances. Normally time proceeds from top to bottom [43]. Message sequence descriptions are provided in sequence diagrams to bring forth meanings of the messages passed between objects. Sequence diagrams describe interactions among software components, and thus are considered to be a good source for cluster level testing. In UML, a message is a request for a service from one UML actor to another, these is typically implemented as method calls. We assume that each sequence diagram represents a complete trace of messages during the execution of a user-level operation.

An example of a UML sequence diagram is shown in Figure 1. The vertical dashed line in the diagram is called a lifeline. A lifeline represents the existence of the corresponding object instance at a particular time. Arrows between the lifelines denote communication between object instances using messages. A message can be a request to the receiver object to perform an operation(of the receiver). A synchronous message is shown with a filled arrowhead at the end of a solid line. An asynchronous message is depicted with an open arrowhead at the end of a solid line. Return messages are usually implied. We can explicitly show return messages using an open stick arrowhead with a dashed line as shown in Figure 1. An object symbol shown with a rectangle is drawn at the head of the lifeline. An activation (focus of control) shows the period during which an instance is performing a procedure. The procedure being performed may be labeled in text next to the activation symbol or in the margin.

UML 2.0 also allows an element called note, for adding additional information to the sequence diagram. Notes are shown with dog-eared rectangle symbols linked to object lifeline through a dashed line as shown in Figure 1. Notes are

convenient to include pseudocode, constraints, pre-conditions, post-conditions, text annotations etc. in sequence diagram. However, in our approach we restrict the notes to contain only executable statements. Messages in the sequence diagram are chronologically ordered. So we have numbered them based on their timestamps. Further, we have numbered the notes in an arbitrary manner.

In UML 2.0, a set of interactions can be framed together and can be reused at other locations. Different interaction fragments can be combined to form a combined fragment. A combined interaction fragment defines an expression of interaction fragments. A combined interaction fragment is defined by an interaction operator and corresponding interaction operands. Through the use of combined fragments, the user will be able to describe a number of traces in a compact and concise manner. A combined fragment with an operator *alt* (for alternative) is shown in Figure 1.

3. Basic Concepts

In this section, we discuss a few basic concepts that are useful to understand the rest of this paper.

Class, Cluster and System Level Testing: In object oriented systems, generally testing is done at different levels of abstraction: class level, cluster level and system level [9, 49, 28]. Class level testing tests the code for each operation supported by a class as well as all possible method interactions within the class. Class level testing also includes testing the methods in each of the states that a corresponding object may assume. At cluster level testing, the interactions among cooperating classes are tested. This is similar to integration testing. The system level testing is carried out on all the clusters making up the complete system.

Executable UML: Executable UML [39, 46] allows model specifications to be efficiently translated into code. Executable UML can formalize requirements and use cases into a rich set of verifiable diagrams. The models are executable and testable and can be translated directly into code by executable UML model compilers. The benefits of this approach go well beyond simply reducing or eliminating the coding stage; it ensures platform independence, avoids obsolescence (programming languages may change, the model doesn't) and allows full verification of the models by executing them in a test and debug environment.

Test Case: A test case is the triplet (I, D, O) where I is the state of the system at which the test data is input, D is the test data input to the system, and O is the expected output of the system [2, 38, 42]. The output produced by the execution of the software with a particular test case provides a specification of the actual software behavior.

4. Dynamic Slicing based Test Case Generation from Sequence Diagrams

In this section we describe our proposed methodology for automatic test case generation from UML sequence diagrams using dynamic slicing. We first define a few terms and the relevant test coverage criteria.

4.1. Definitions

The following definitions would be used in the description of our methodology.

Message Dependency Graph (MDG): We define MDG as a directed graph with (N, E), where N is a set of nodes and E is a set of edges. MDG shows the dependency of a given node on the others. Here a node represents either a message or a note in the sequence diagram and edges represent either control or data dependency among nodes. Here we have assumed that notes are attached to objects and the statements on the notes are executed when its corresponding lifeline is activated. MDG does not distinguish between control or data dependence edges. It does however distinguish between stable and unstable edges. Definitions of stable and unstable edges are given subsequently. The induced subgraph

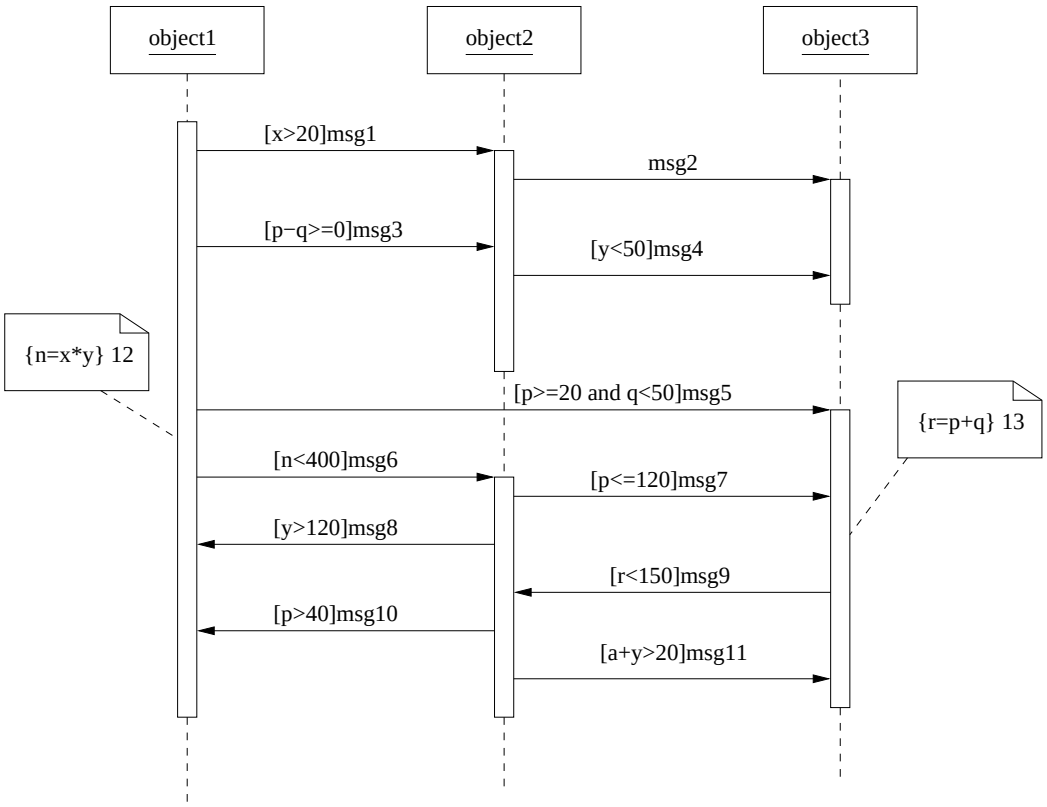


Figure 2. An example sequence diagram

of MDG of the sequence diagram in Figure 2 on the Node Set(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13) is shown in Figure 3. of Subsection 4.7.

Slicing Criteria: Slices are constructed based on a slicing criterion. Weiser's slicing criterion [51] consisted of a set of variables of interest and a point of interest within the original program. Statements which cannot affect the values of variables at a point of interest in the program are removed to form the slice. In our case, the slicing criterion (m, V) specifies the location (identity) of a message m in its corresponding MDG and V is a set of variables that are used by the conditional predicate on the message at m.

Dynamic Slice: A dynamic slice of a sequence diagram is defined with respect to its corresponding MDG. Consider a predicate in MDG on a message m in a sequence diagram. A dynamic slice is the induced subgraph of MDG, induced by the set of nodes in MDG that affect a predicate at m for a given execution. We call this slice as a dynamic slice of sequence diagram. Those nodes of MDG that do not affect the predicate at m are removed to form the slice, for the slicing criterion (m, V).

UseVar(x): It is the set of all nodes in MDG that uses the value of variable x. For example, in the expression $(n = x * y)$ there is a use of the value of the variable x.

AllotVar(x): It is the set of all nodes in MDG that defines the variable x. In addition, consider a conditional guard specifies a condition in a message using variable x. If x is used to specify another condition in another message, such conditional guards are also treated as members of AllotVar(x). We use the term allotment to indicate that a variable x is either defined or if x is used to specify guards in the rest of paper. For example, consider nodes 4 and 8 in MDG as shown in Figure 3. These nodes correspond to messages $[y < 50]msg4$ and $[y > 120]msg8$ respectively in Figure 2. For a particular input value for the variable y, only one of these messages will take place. Hence, both nodes 4 and 8 are treated as members of AllotVar(y). A use of y will require only one of the AllotVar(y), not both.

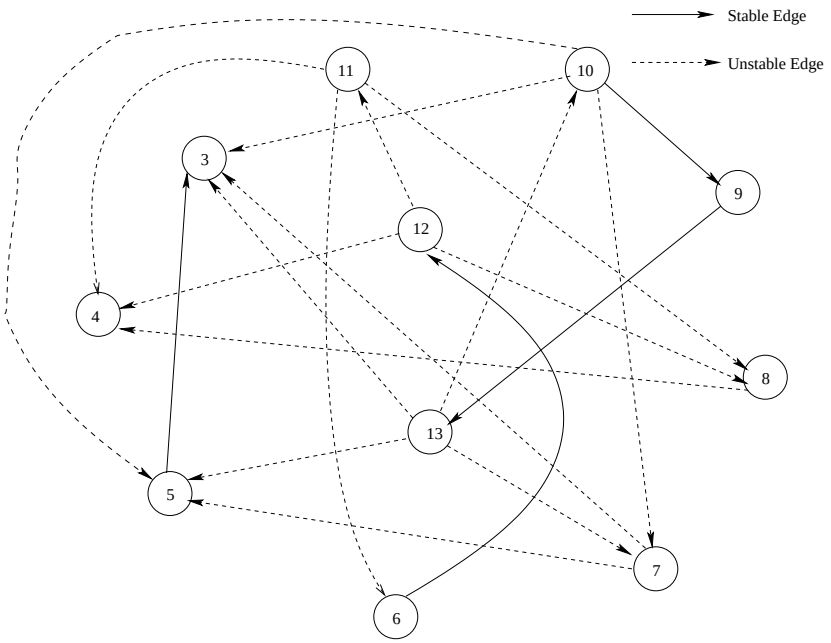


Figure 3. The induced subgraph of dependency graph of the sequence diagram in Figure 2 on the node set (3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

Dependence Edge: If node i is a member of usevar(x) and a node j is a member of AllotVar(x), then there is a directed edge from node i to node j, this edge is also called a dependence edge.

Dependency Path: A dependency path F from some node v_i, to a node v_k is a sequence of nodes and edges in MDG from v_i to v_k.

Unstable Edge: Let M, M_i, M_j be three nodes in MDG. An outgoing dependence edge (M, M_i) in MDG is said to be unstable if there exists an outgoing dependence edge (M, M_j) with M_i not equal to M_j such that the statements M_i and M_j both are members of AllotVar(x). For example, in Figure 2, the messages $[y < 50]msg4$ and $[y > 120]msg8$ will form unstable edges with respect to the message $[a + y > 20]msg11$. These edges corresponds to (4, 11) and (8, 11) respectively in the MDG shown in Figure 3.

Stable Edge: An edge in a dependency graph is said to be stable, if it is not an unstable edge.

Slice Condition: Consider a slice S of a sequence diagram for the slicing criterion (m, V). The slice condition of the slice S is the conjunction of all the individual predicates present in the dynamic slice for a given execution.

Slice Domain: The slice domain of slice S is the set of all input data values for which the slice condition of S is satisfied.

Boundary: A slice domain is surrounded by a boundary. A boundary is a set of data points. A boundary might consist of several segments and each segment of the boundary is called a border [16]. Each border is determined by a single simple predicate in the slice condition. A border crossing occurs for some input where the conditional predicate changes its Boolean value from true to false or vice versa.

4.2. Test Coverage

A software test data adequacy criterion (or coverage criterion) is used to find out whether a set of test cases is sufficient, or "adequate", for testing a given software. Some of the relevant test criteria are introduced in this section.

4.2.1. Slice Coverage Criterion

Several test coverage criteria such as message path criteria, full predicate coverage etc., have been proposed in the literature [2]. Several other criteria such as slice coverage criterion can easily be formulated based on these criteria. We extend

slice coverage criterion from path based criteria. Slice criteria is defined with respect to a dependency graph. We define slice coverage criterion for sequence diagram as follows: Consider a test set T and an MDG corresponding to a sequence diagram SD. In order to satisfy the slice coverage criterion, it is required that T must cause all the dependency paths in MDG for each slice to be taken at least once. Slice coverage ensures that all the dependency paths of an MDG (Message Dependency Graph) are covered.

4.2.2. Full Predicate Coverage

Full predicate coverage criterion requires that each clause should be tested independently [42]. In other words, each clause in each predicate on every message must independently affect the outcome of the predicate. Given a test set T and sequence diagram SD, T must cause each clause in every predicate on each message in SD to take on the values of TRUE and FALSE while all other clauses in the predicate have values such that the value of the predicate will always be the same as the clause being tested. This ensures that each clause in a condition is separately tested.

4.2.3. Boundary Testing Criterion

Testers have frequently observed that domain boundaries are particularly fault-prone and should therefore be carefully checked [24]. Boundary testing criterion is applicable whenever the test input domain is subdivided into subdomains by decisions (conditional predicates). Let us select an arbitrary border for each predicate p. We assume that the conditional predicates on the sequence diagram are relational expressions (inequalities). That is, all conditional predicates are of the following form: $E_1 op E_2$, where $E_1 and E_2$ are arithmetic expressions, and op is one of $\{<, \leq, >, \geq\}$. Jeng and Weyuker [24] have reported that an inequality border can be adequately tested by using only two points of test input domain, one named ON point and the other named OFF point. The ON point can be anywhere on the given border. It does not even have to lie exactly on the given border. All that is necessary is that it satisfies all the conditions associated with the border. The requirement for the OFF point is that it be as close to the ON point as possible, but it should lie outside the border.

The boundary testing criterion can now be defined as follows: The boundary testing criterion is satisfied for inequality borders if each selected inequality border b is tested by two points (ON-OFF) of test input domain such that, if for one of the points the outcome of a selected predicate q is true, then for the other point the outcome of q is false. Also the points should satisfy the slice condition associated with b and the points should be as close as possible to each other [16].

Definitions of boundary testing criteria for equality and non-equality borders are defined in [24, 16]. For conciseness, we do not consider them here. However they can easily be considered in our approach. We use boundary testing as an extension of slice coverage criterion. The number of test cases to be generated for achieving slice coverage criterion can be very large if we use a random approach. We reduce this by using boundary testing along with slice coverage. For example consider a the predicate $(n < 400)$ shown in Figure 2. In the example section we have shown two test data that can be used to test it and they are [21, 20] and [21, 19] where, the test data has the values of $[x, y]$ for the predicate $(n < 400)$. The slice condition consists of $(x > 20), (y < 50), (n = x * y)$ and the test data is generated subject to slice condition. Instead of generating a set of test cases randomly and selecting the test cases from this set that satisfies this slice condition, we generate two test cases based on a simple predicate using boundary testing.

4.3. Overview of Our Approach

In our approach, the first step is to select a conditional predicate on the sequence diagram. The order in which we select predicates is the chronological order of messages appearing in a sequence diagram. For each message in the sequence diagram, there will be a corresponding node in the MDG. For each conditional predicate, we create

the dynamic slice for the slicing criteria (m, V) and with respect to each slice we generate test data. The generated test data for each predicate corresponds to the true or false values of the conditional predicate and these values are generated subject to the slice condition. This helps to achieve slice coverage. The different steps of our approach are elaborated in the following subsections.

4.4. Dynamic Slice of Sequence Diagrams

In our approach, a dynamic slice of a sequence diagram is constructed from its corresponding MDG (Message dependency graph). An MDG is created statically and it needs to be created only once. For each message in the sequence diagram, there will be a corresponding node in the MDG. From MDG, we create the dynamic slice corresponding to each conditional predicate, for the slicing criteria (m, V). For creating dynamic slices we use an edge marking method. Edge marking methods are reported in [22, 40] for generating dynamic slices in the context of procedural programs. Their edge marking methods uses program dependence graph. We generate a message dependence graph from UML sequence diagram and apply the edge marking technique on it. Edge marking algorithm is based on marking and unmarking the unstable edges appropriately as and when dependencies arise and cease at run time. After an execution of the node x at run-time, an unstable edge(x, y) is marked if the node x uses the value of the variable v at node y and node y is a member of AllotVar(v). A marked unstable edge(x, y) is unmarked after an execution of a node z if the nodes y and z are in AllotVar(v), and the value of v computed at node y does not affect the present value of v at node z. In our approach we generate test data that satisfies all constraints corresponding to a slice.

Before execution of a message sequence M, the type of each of its edges in MDG is appropriately recorded as either stable or unstable. The dependence associated with a stable edge exists at every point of execution. The dependence associated with an unstable edge keeps on changing with the execution of the node. We mark an unstable edge when its associated dependence exists, and unmark when its associated dependence ceases to exist. Each stable edge is marked and each unstable edge is unmarked at the time of construction of the MDG. We mark and unmark edges during the execution of the message sequences, as and when a dependencies arise or cease, and a stable edge is never unmarked. Let dslice(n) denote the dynamic slice with respect to the most recent execution of the node n. Let $(n, x_1), (n, x_2), \ldots, (n, x_n)$ be all the marked outgoing dependence edges of n in the updated MDG after an execution of the node n. It is clear that the dynamic slice with respect to the present execution of the node is $dslice(n) = x_1, x_2, \ldots, x_n \cup dslice(x_1) \cup \cdots \cup dslice(x_n)$.

We now present the edge marking dynamic slicing algorithm for sequence diagrams in pseudocode form. Subsequently this method is explained using an example.

Edge Marking Dynamic Slicing Algorithm for Sequence Diagrams

- Do before execution of the message sequence:-
  - Unmark all the unstable edges.
  - Set dslice(n) = NULL for every node n of the MDG.
- For each node n of the message sequence Do
  - For every variable used at n, mark the unstable edge corresponding to its most recent allotment. (Suppose there is a predicate $x > 50$ which is true for the given execution step and inputs then the edge to that predicate is marked. If the predicate is false then it remains unmarked.)
  - Update dslice(n).
  - If n is a member of AllotVar(x) and n is not a UseVar(x) node, then do the following:-
    - Unmark every marked unstable edge (n_1, n_2) with $n_1 \in UseVar(x)$ and n_2 is a node that does not affect the present allotment of the variable var. Hence, the marked unstable edge (n_1, n_2) representing the dependence of node n_1 on node n_2 in the previous execution of node n_1 will not continue to represent the same dependence in the next execution of node n_1.

For example, let x = 30, y = 45, p = 55, q = 40, a = 10 be a data set for the diagram given in Figure 2. For the slicing criteria (11, y), initially let edges (11, 4) and (11, 8) are unmarked unstable edges as seen in Figure 3. During the execution of node 11, for the given data set, we mark the unstable edge (11, 4) whereas the unstable edge (11, 8) remains unmarked as the value of y at present is 45. Hence the dynamic slice of node 11 for the slicing criteria (11, y) is $4 \cup dslice6$ and do not include 8. Let at some other execution, the data set is x = 30, y = 55, p = 45, q = 40, a = 10. In this case the dynamic slice of node 11 for the slicing criteria (11, y) is $8 \cup dslice6$ and do not include 4.

4.5. Generation of Predicate Function

Consider an initial set of data I_0 that is randomly generated for the variables that affect a predicate p in a slice S. As already mentioned in our approach, we compute two points named ON and OFF for a given border satisfying the boundary testing criterion. We transform the relational expressions of the predicates to a function F (Predicate Function). If the predicate p is of the form (E1 op E2), where E1 and E2 are arithmetic expressions, and op is a relational operator, then F = (E1 – E2) or (E2 – E1) depending on whichever is positive for the data I_0. Next we successively modify the input data I_0 such that the function F decreases and finally turns negative. When F turns negative, it corresponds to the alternation of the outcome of the predicate. Hence as a result of the above predicate transformation, the change in the outcome of predicate p now corresponds to the problem of minimization of the function F. This minimization can be achieved through repeated modification of input data value.

4.6. Test Data Generation

The basic search procedure we use for finding the minimum of the predicate function F is the alternating variable method [29, 16] which consists of minimizing F with respect to each input variable in turn. Each input data variable x_i is increased/decreased in steps of Ux_i, while keeping all the other data variables constant. Here Ux_i refers to a unit step of the variable x_i. The unit step depends on the data type being considered. For example, the unit step is 1 for integer values. The method works with many other types of data such as float, double, array, pointer etc. However the method may not work when the variable assumes only a discrete set of values. Each predicate in the slice can be considered to be a constraint. If any of the constraint is not satisfied in the slice, for some input data value, we say that a constraint violation has taken place. We compute the value of F when each input data is modified by Ux_i. If the function F has decreased on the modified data, and constraint violation has not occurred, then the given data variable and the appropriate direction is selected for minimizing F further. Here appropriate direction refers to whether we increase or decrease the data variable x_i. We start searching for a minimum with an input variable while keeping all the other input variables constant until the solution is found (the predicate function becomes negative) or the positive minimum of the predicate function is located. In the latter case, the search continues from this minimum with the next input variable.

4.7. An Example

Consider an example sequence diagram as shown in Figure 2. We have selected this example as it demonstrates the concepts in our approach. We illustrate our methodology by explaining the test data generation for the predicate $(n < 400)$ shown in Figure 2. Its corresponding MDG is shown in Figure 3. Let the slicing criterion be (6, n). For this slicing criterion, the slice contains of the set of nodes that corresponds to predicates $(x > 20), (y < 50), (n = x * y)$. The function F will be the expression $(n - 400)$. Let I_0 be the initial data: [25, 40] where (x = 25, y = 40). The condition $(n < 400)$ is false for I_0 as $(1000 < 400)$. The function F will be the expression $(n - 400)$ and $F(I_0) = 600$. We should minimize F, in order to alter the boolean outcome of predicate $(n < 400)$, which is false initially.

First we decrease the value of data x in steps. In the first step, we take x = 24 and the value of F is calculated as 560 for [x, y] = [24, 40]. Observe that the function F reduces by reducing x. Therefore in the next step, the size of the step is doubled and hence the value of variable x is decreased by 2. As we minimize F further in several iterations, we finally arrive at two data points with [x, y] = [21, 20], F is positive and the condition ($n < 400$) is still false. So we take two data sets I_in as [x, y] = [21, 20] that makes F positive (or zero) and another data set I_out as [x, y] = [21, 19] that makes F negative.

The test cases we generate for the predicate ($n < 400$) are (object1, [21, 19], object2) and (object1, [21, 20], object1) correspond to different truth values of the predicate ($n < 400$). Here test cases has the form (sender object, [test data], receiver object). Test data has the values of [x, y] for the predicate ($n < 400$). These test cases are generated satisfying the slice condition of the slice. With our proposed method we generate test cases for each such conditional predicates on the sequence diagram.

5. An Implementation

To the best of our knowledge, no full-fledged ready made tool exists that are publicly available to execute UML models. Hence, for generating dynamic slices in our experimentation, we have simulated the executions. We made a prototype tool that implements our method. Figure 4 shows the important classes that we used to generate test cases from sequence diagram in our implementation. *SliceGenerator* class creates the message dependency graph. It makes the sets defSet and useSet for each variable in the sequence diagram. It forms slices based on the slicing criteria for each of the messages in the sequence diagram. *SliceRecord* class keeps a record of slices.

DocumentParser class parses the XML file corresponding to a UML sequence diagram. We used the Document Object Model (DOM) API that comes with the standard edition of the Java platform, for parsing XML files. The package org.w3c.dom.*, provides the interfaces for the DOM. The DOM parser begins by creating a hierarchical object model of the input XML document. This object model is then made available to the application for it to access the information it contains in a random access fashion. This allows an application to process only the data of interest and ignore the rest of the document.

XmlBoundary is the class of the program from which the execution starts. It accepts an XML file of sequence diagram from a user. Then it extracts the parent tag of the XML file and passes the tag (called head) to the TestCaseController class. *TestCaseController* class coordinates the different activities of the program. *TestCaseBoundary* class is responsible for displaying the list of test cases for a collaboration diagram. The source and destination objects as well as the slice condition is printed along with test data.

In our prototype implementation, we have considered only integer and Boolean variables as part of the conditional expression in sequence diagrams. Other data types however can easily be considered. Further, for the prototype implementation we have assumed that the necessary constraints are available in notes instead of class/object diagrams. Extracting data types of attributes, or constraints from class/object diagrams for our implementation can be easily done. The GUI was developed using the swing component of Java. A GUI screen along with a sample sequence diagram is shown in Figure 5. The GUI gives the flexibility to view the sequence diagram, its XML representation and the generated test cases. Figure 6 shows the UTG display of the XML file of example given in Figure 5. The corresponding test cases generated are shown in Figure 7. Our tool allows storing the test cases as text files for later processing.

We have implemented our method for generating test cases automatically from UML sequence diagrams in a prototype tool named UTG. Here, UTG stands for UML behavioral Test case Generator. UTG has been implemented using Java and can easily integrate with any UML CASE tools like MagicDraw UML [41] that supports XML (Extensible Markup Language) format. Since UTG takes UML models in XML format as input,

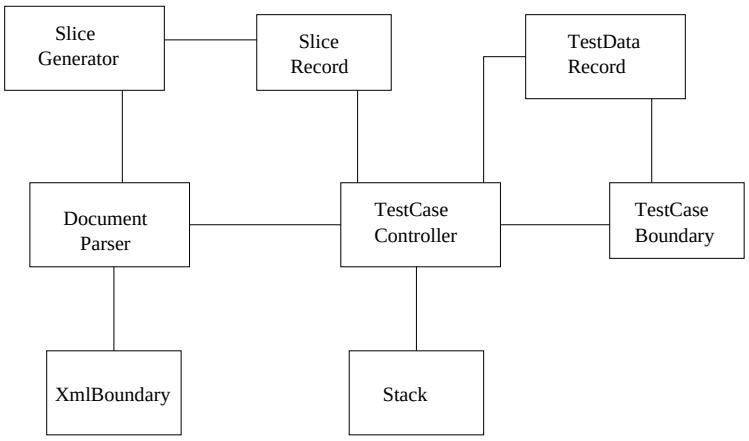


Figure 4. Class diagram of UTG for generating test cases from sequence diagrams

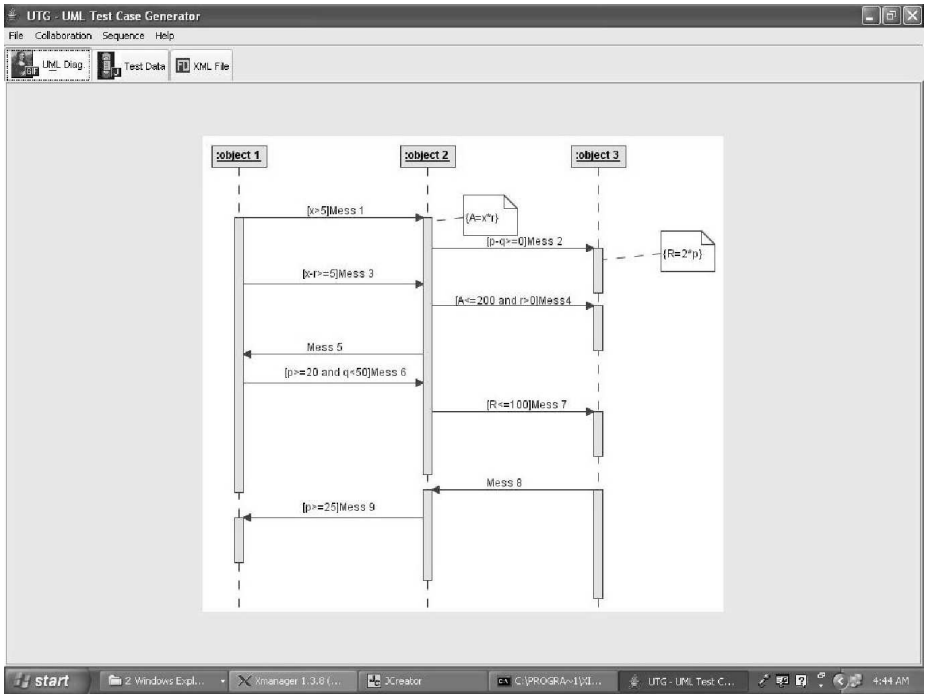


Figure 5. The GUI screen of UTG with an example sequence diagram

UTG is independent of any specific CASE tool. We have used the tool with several UML designs and the tool was found effective in generating test cases. The generated test cases were found to achieve the desired coverage.

6. Related Work

Bertolino and Basanieri [4] proposed a method to generate test cases following the sequence of messages between components in a sequence diagram. They develop sequence diagrams for each use case and use category partition method to generate test data. They characterize a test case as a combination of all suitable choices of the involved settings and interactions in a sequence of messages. In another interesting work, Basanieri, et al. [3] describe the CowSuite approach which provides a method to derive the test suites and a strategy for test prioritization and selection. CowSuite is mainly based on the analysis of the use case diagrams and sequence diagrams. From these two diagrams they construct a graph structure which is a mapping of the project architecture and this graph is ex-

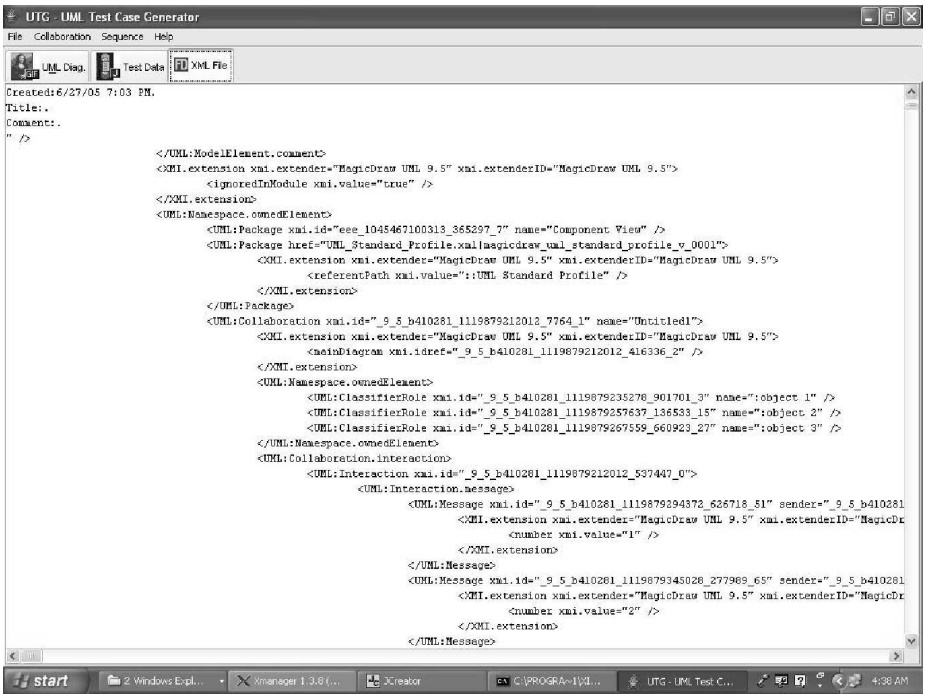


Figure 6. A screen shot of UTG with a portion of the XML file corresponding to example in Figure 5

plored using depth-first search algorithm. They use category partition method [45] for generating test cases. They construct test procedures using the information retrieved from the UML diagrams.

Briand and Labiche [7] describe the TOTEM (Testing Object-orienTed systEms with the Unified Modeling Language) system test methodology. Functional system test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and from OCL used in all these artifacts. They represent sequential dependencies among use cases by means of an activity diagram constructed for each actor in the system. The derivation of use case sequences from the activity diagram is done with a depth first search through a directed graph capturing the activity diagram. They generate legal sequences of use cases according to the sequential dependencies specified in the activity diagram. Abdurazik and Offutt [1] proposed novel and useful test criteria based on collaboration diagrams for static checking and dynamic testing based on collaboration diagrams. They recommended a criterion for dynamic testing that involved message sequence paths. They adapt traditional data flow coverage criteria (eg. all definition – uses) in the context of UML collaboration diagrams.

Linzhang, et al. [34] proposed a gray-box testing method using UML activity diagrams. They propose an algorithm to generate test scenarios from activity diagrams. The information regarding input/output sequence, parameters, the constraint conditions and expected object method sequence is extracted from each test scenario. They recommend applying category-partition method to generate possible values of all the input/output parameters to find the inconsistency between the implementation and the design.

Among all UML diagrams, test case generation from state chart diagram has possibly received maximum attention from researchers [8, 21, 27, 28, 42, 48]. Offutt and Abdurazik [42] developed an interesting technique for generating test cases from UML state diagrams which is intended to help perform class-level testing. Their method takes a state transition table as input, and generates test cases for the full predicate coverage criterion. It processes each outgoing transition of each source state, generates a test case that makes the transition taken, and then generates test cases that make the transition untaken. A test case is designed corresponding to

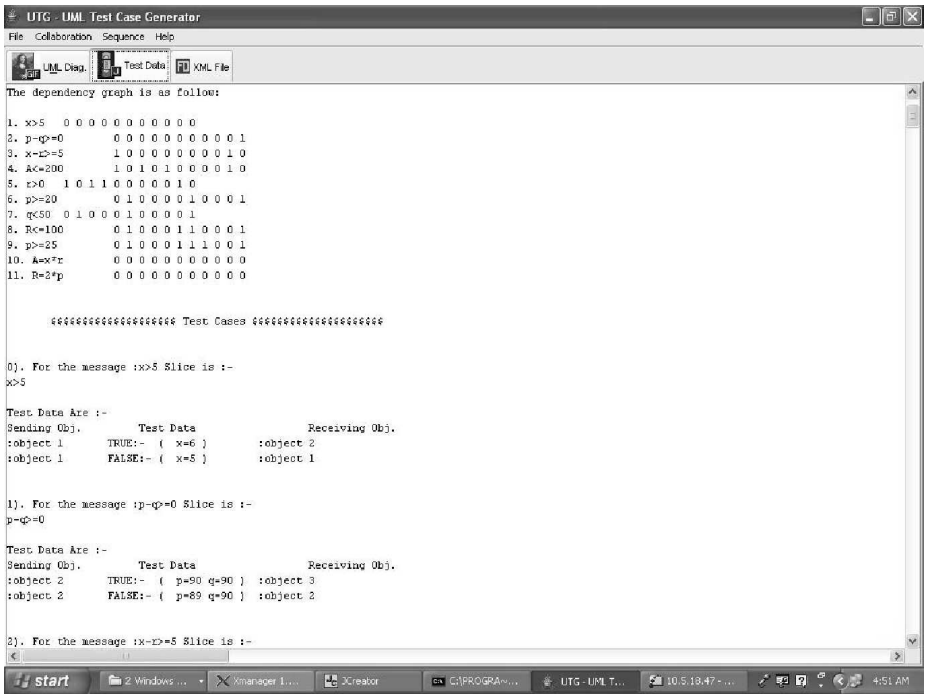


Figure 7. A screen shot of UTG with dependency graph and generated test cases corresponding to the example of Figure 5

each variable in a transition predicate. To avoid redundant test case value assignments, those variables that have already been assigned values are not considered in the subsequent test case value assignment process. After all test case values are generated, an additional algorithm is run on the test cases to identify and remove redundant test cases. Kansomkeat, et al. [27] have proposed an alternate method for generating test sequences using UML state chart diagrams. They transform the state chart diagram into an intermediate diagram called Testing Flow Graph (TFG) which is used to generate test sequences. TFG is a flattened hierarchy structure of states. The testing criterion they proposed is the coverage of states and transitions of TFG.

Kim, Y.G. et al. [28] proposed a method for generating test cases for class testing using UML state chart diagrams. They transform state charts to extended finite state machines (EFSMs) to derive test cases. The hierarchical and concurrent structure of states is flattened and broadcast communications are eliminated in the resulting EFSMs. Next data flows are identified by transforming EFSMs into flow graphs to which conventional data flow analysis techniques are applied. Hartmann et al. [21] augment the UML description with specific notations to create a design-based testing environment. The developers first define the dynamic behavior of each system component using a state diagram. The interactions between components are then specified by annotating the state diagrams, and the resulting global FSM that corresponds to the integrated system behavior is used to generate the tests.

Scheetz et al. [48] developed an approach for generating system (black box) test cases using an AI (Artificial Intelligence) planner. They used UML class diagrams and state diagrams to represent the conceptual architecture of a system under test. They developed a representation method at the application domain level that allows statement of test objectives at that level, and their mapping into a planner representation. Their method maps the initial and goal conditions into a problem description for the planner. The planner generates a plan based on this input. In the next step, they carry out a conversion of the plan to produce executable test cases. The purpose of a test case in a goal directed view is to try to change the state of the

overall system to the goal state. The planner decides which operators will best achieve the desired goal states. Cavarra, et al. [8] use UML class diagrams, state diagrams, and object diagrams to characterize the behavior of a system. These UML diagrams are translated into formal behavioral descriptions, written in a language of communicating state machines and used as a basis for test generation. From this they form a test graph, consisting of all traces leading to an accept state, together with branches that might lead to invalid state.

Andrews et al. [2] describe several useful test adequacy criteria for testing executable forms of UML. The criteria proposed for class diagrams include association-end multiplicity criterion, generalization criterion and class attribute criterion. The interaction diagram criteria like condition coverage, full predicate coverage, each message on link, all message paths and collection coverage criteria are used to determine the sequences of messages that should be tested. They also describe a test process. Ghosh et al. [15] present a testing method in which executable forms of Unified Modeling Language (UML) models are tested. In systematic design testing, executable models of behaviors are tested using inputs that exercise scenarios. This can help reveal flaws in designs before they are implemented in code. Their method incorporates the use of test adequacy criteria based on UML class diagrams and interaction diagrams. Class diagram criteria are used to determine the object configurations on which tests are run, while interaction diagram criteria are used to determine the sequences of messages that should be tested. These criteria can be used to define test objectives for UML designs. Engels et al. [12] discuss how consistency among different UML models can be tested. They propose dynamic meta modeling rules as a notation for the consistency conditions and provide the concept for an automated testing environment using these rules.

In contrast with the above discussed approaches we generate actual test cases from sequence diagrams. Our approach can work on executable forms of UML design specifications and is meant for cluster level testing where object interactions are tested. Corresponding to each conditional predicate on the sequence diagram, we construct dynamic slice from its MDG and with respect to the slice we generate test data. Our test data generation scheme is automatic.

Kamkar et al. [26] explains how interprocedural dynamic slicing can be used to increase the reliability and precision of interprocedural data flow testing. Harman and Danicic [18] presents an interesting work that illustrates how slicing will remove statements which do not affect a program variable at a location thereby simplifying the process of testing and analysis. They also provide a program transformation algorithm to make a program robust. Slicing has been used as a reduction technique on specifications like state models [32]. Anyhow this work [32] do not provide a scheme for test generation.

Korel [29] generated test data based on actual execution of a program under test. He used function minimization methods and dynamic data flow analysis. If during a program run an undesirable execution flow is observed (e.g., the "actual" path does not correspond to the selected control path), then function minimization search algorithms are used to automatically locate the values of input variables for which the selected path is traversed. This helps in achieving path coverage. In addition, dynamic data flow analysis is used to determine those input variables that are responsible for the undesirable program behavior, leading to significant speedup of the search process. Hajnal et al. [16] extended the work done by Korel [29]. They reported the use of boundary testing that requires the testing of one border only along a selected path. The test input domain may be surrounded by a boundary and each segment of the boundary is called a border. The task to generate two test data points considering only one border for each path, is much easier. Their testing strategy can also handle compound predicates. Jeng and Weyuker [24] have reported that an inequality border can be tested by only two points of test input domain, one named ON point and another named OFF point. For borders in a discrete space containing no points lying exactly on the border, their strategy allows the ON point to

be chosen from beneath the border as long as the distance between the ON and OFF points is minimized. These works [26, 18, 29, 16, 24] discussed above have focused on unit testing of procedural programs.

7. Conclusion

We have presented a novel method to generate test cases by dynamic slicing UML sequence diagrams. Our approach is meant for cluster level testing where object interactions are tested. Our approach automatically generates test data, which can be used by a tool to carry out automatic testing of a program. Generation of MDG is the only static part in our approach. We identify the conditional predicates associated with messages in a sequence diagram and create dynamic slice with respect to each conditional predicate. We generate test data with respect to each constructed slice and the test data is generated satisfying slice condition. We have formulated a test adequacy criterion named slice coverage criterion. We have implemented our methodology to develop a prototype tool which was found effective in generating test cases. The test cases generated can also be used for conformance testing of the actual software where the implementation is tested to check whether it conforms to the design. The slicing approach was found to be especially advantageous when the number of messages in the sequence diagram is large. We need to consider only the slices for finding test cases instead of having to look at the whole sequence diagram. If the sequence diagram is large it becomes very complex and difficult to find test cases manually. If we know where to look for errors it becomes a great simplification and saves a lot of time and resources. The slices help to achieve this simplification. The generated test cases were found to achieve slice coverage.

Acknowledgements. The authors would like to thank Pratyush Kanth and Sandeep Sahoo for implementing our approach presented in this paper.

References

[1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd International Conference on the UML, Lecture Notes in Computer Science*, volume 1939, pages 383–395, York, U.K., October 2000. Springer-Verlag GmbH.

[2] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing Verification and Reliability*, 13:97–127, 2003.

[3] F. Basanieri, A. Bertolino, and E. Marchetti. The cow suit approach to planning and deriving test suites in UML projects. In *Proceedings of the Fifth International Conference on the UML, LNCS*, volume 2460, pages 383–397, Dresden, Germany, October 2002. Springer-Verlag GmbH.

[4] A. Bertolino and F. Basanieri. A practical approach to UML-based derivation of integration tests. In *Proceedings of the 4th International Software Quality Week Europe and International Internet Quality Week Europe*, Brussels, Belgium, 2000. QWE.

[5] R.V. Binder. Testing object-oriented software: a survey. *Software Testing Verification and Reliability*, 6(3/4):125–252, 1996.

[6] D. Binkley and K. Gallagher. *Program Slicing*, volume 43 of *Advances in Computers*. Academic Press, 1996.

[7] L. Briand and Y. Labiche. A UML-based approach to system testing. In *Proceedings of the 4th International Conference on the UML, LNCS*, volume 2185, pages 194–208, Toronto, Canada, January 2001. Springer-Verlag GmbH.

[8] A. Cavarra, C. Crichton, and J. Davies. A method for the automatic generation of test suites from object models. *Information and Software Technology*, 46(5):309–314, 2004.

[9] H.Y. Chen, T.H. Tse, and T.Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(4):56–109, January 2001.

[10] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *IEEE International Conference on Software Maintenance (ICSM'97)*, pages 188–195. IEEE Computer Society Press, Los Alamitos, USA, 1997.

[11] T.T. Dinh-Trong. Rules for generating code from UML collaboration diagrams and activity diagrams. Master's thesis, Colorado State University, Fort Collins, Colorado, 2003.

[12] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proceedings of the Sixth International Conference on Integrated Design and Process Technology(IPDT)*, USA, 2002. Society for Design and Process Science.

[13] G. Engels, R. Hucking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformations to Java. In *Proceedings of the 2nd International Conference on the UML, LNCS*, volume 1723, pages 473–488, Berlin / Heidelberg, October 1999. Springer.

[14] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[15] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 332–343. IEEE Computer Society, November 2003.

[16] A. Hajnal and I. Forgacs. An applicable test data generation algorithm for domain errors. In *ACM SIGSOFT Software Engineering Notes, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, volume 23, 1998.

[17] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[18] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing Verification and Reliability*, 5(3):143–162, September 1995.

[19] M. Harman, C. Fox, R.M. Hierons, D. Binkley, and S. Danicic. Program simplification as a means of approximating undecidable propositions. In *7th IEEE Workshop on Program Comprehension*, pages 208–217. IEEE Computer Society Press, Los Alamitos, USA, 1999.

[20] M. Harman and K.B. Gallagher. Program slicing. *Information and Software Technology*, 40:577–581, December 1998.

[21] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *ACM SIGSOFT Software Engineering Notes, Proceedings of International Symposium on Software testing and analysis*, volume 25, August 2000.

[22] J. Horgan and H. Agrawal. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, volume 25, pages 246–256, White Plains, New York, 1990. SIGPLAN Notices, Analysis and Verification.

[23] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[24] B. Jeng and E.J. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(3), 1994.

[25] P.C. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9), September 1994.

[26] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance*, pages 386–395. IEEE Computer Society, Washington, DC, USA, 1993.

[27] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In *Proceedings of SAICSIT 2003*, pages 296–300. ACM, 2003.

[28] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test cases generation from UML state diagrams. *Proceedings: Software*, 146(4):187–192, 1999.

[29] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[30] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[31] B. Korel and J. Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40:647–659, 1998.

[32] B. Korel, I. Singh, L.H. Tahat, and B. Vaysburg. Slicing of state-based models. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 34–43. IEEE, 2003.

[33] A. Lakhotia and J.C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology Special Issue on Program Slicing*, 40:677–689, 1998.

[34] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating test cases from UML activity diagrams based on gray-box method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 284–291. IEEE, 2004.

[35] A. De Lucia. Program slicing: methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE, November 2001.

[36] A. De Lucia, A.R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In *4th IEEE Workshop on Program Comprehension*, pages 9–18. IEEE Computer Society Press, Los Alamitos, USA, 1996.

[37] J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *2nd International Conference on Computers and Applications*, pages 877–882. IEEE Computer Society Press, Los Alamitos, USA, 1987.

[38] R. Mall. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2003.

[39] S.J. Mellor and M.J. Balcer. *Executable* UML*: A Foundation for Model Driven Architecture*. Addison-Wesley: Reading, MA, 2002.

[40] G.B. Mund, R. Mall, and S. Sarkar. An efficient program slicing technique. *Information and Software Technology*, 44:123–132, 2002.

[41] No Magic Inc. *MagicDraw* UML*, Version 9.5*, Golden, CO, www.magicdraw.com.

[42] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on UML, Lecture Notes in Computer Science*, volume 1723, pages 416–429, Fort Collins, TX, 1999. Springer-Verlag GmbH.

[43] OMG. *Unified Modeling Language Specification, Version 2.0*. Object Management Group, www.omg.org, August 2005.

[44] L. Osterweil. Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, 28(4), December 1996.

[45] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6), June 1998.

[46] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable* UML. Cambridge University Press, 2004.

[47] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a uml virtual machine. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 36, pages 327–341. ACM SIGPLAN Notices, ACM Press, USA, October 2001.

[48] M. Scheetz, von A. Mayrhauser, and R. France. Generating test cases from an object oriented model with an AI planning system. In *Proceedings of the 10th International Symposium on Software Reliability Engineering, ISSRE 99*, pages 250–259. IEEE Computer Society Press, 1999.

[49] M.D. Smith and D.J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53, June 1992.

[50] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, June 1995.

[51] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[52] C.E. Williams. Software testing and the UML. In *Proceedings of the International Symposium on Software Reliability Engineering, (ISSRE'99)*, Boca Raton, FL, November 1999.

http://www.e-informatyka.pl/wiki/e-Informatica

e-Informatica

ISSN 1897-7979