e-Informatica

software engineering journal

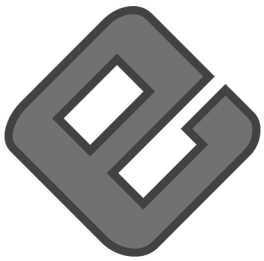
2008 volume 2 issue 1

e-Informatica

e-Informatica

software engineering journal

2009 | volume 3 | issue 1



e-Informatica

Wrocław University of Technology

Editors

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

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

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

Printed in the camera ready form

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

ISSN 1897-7979

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Order No. 757/2009 .

Editorial Board

Contents

Editorial

It is a pleasure to present to our readers the third issue of the e-Informatica Software Engineering Journal (ISEJ).

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

The third issue of the journal includes nine papers. Eight of the papers are extended versions of the best papers presented at the CEE-SET'2008 conference (IFIP Central and Eastern European Conference on Software Engineering Techniques) carefully selected by the editors, while the ninth is a regular paper.

The first of the papers by Ionita et al. presents how domain modelling may leverage the hierarchical composition, supporting two orthogonal mechanisms for composing completely autonomous parts. The vertical mechanism is in charge of coordinating heterogeneous components, tools or services at a high level of abstraction, by hiding the technical details. The result of such a composition is called "domain" and is characterised by a Domain Specific Language (DSL). The horizontal mechanism composes domains at the level of their DSLs, even if they have been independently designed and implemented. The second paper by Vrani et al. describes the approach to aspect-oriented change realization based on a two-level change type model in the web application domain. The third paper by Nikiforova proposes two hemisphere model driven approach for generation of UML class diagram. The fourth paper by Franců and Hnětynka presents an approach that allows automated generation of executable code directly from the use cases written in a natural language. The fifth paper by Majtás presents tool based support of the pattern instance creation on the model level in a semi automatic way. The sixth paper by Ratkowski et al. demonstrates a transformational approach to the design of executable processes in Business Process Execution Language (BPEL). The seventh paper by Winter and Rönkkö is about balancing agile and formal usability test results. The eight paper by Samolej and Szmuc focuses on a new software tool for web-server systems development. The tool consist of a set of predefined Hierarchical Timed Coloured Petri Net (HTCPN) structures – patterns. The last paper by Rychlý is a regular one and presents the component model that addresses component mobility including dynamic reconfiguration, allows to combine control and functional interfaces, and separates a component's specification from its implementation.

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 3, Issue 1, 2009

A Component Model with Support of Mobile Architectures and Formal Description

Marek Rychlý*

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

rychly@fit.vutbr.cz

Abstract

Common features of current information systems have significant impact on software architectures of these systems. The systems can not be realised as monoliths, formal specification of behaviour and interfaces of the systems' parts are necessary, as well as specification of their interaction. Moreover, the systems have to deal with many problems including the ability to clone components and to move the copies across a network (component mobility), creation, destruction and updating of components and connections during the systems' run-time (dynamic reconfiguration), maintaining components' compatibility, etc. In this paper, we present the component model that addresses component mobility including dynamic reconfiguration, allows to combine control and functional interfaces, and separates a component's specification from its implementation. We focus on the formal basis of the component model in detail. We also review the related research on the current theory and practice of formal component-based development of software systems.

1. Introduction

Increasing globalisation of information society and its progression create needs for extensive and reliable information technology solutions. Common requirements for current information systems include adaptability to variable structure of organisations, support of distributed activities, integration of well-established (third party) software products, connection to a variable set of external systems, etc. Those features have significant impact on software architectures of the systems. The systems can not be realised as monoliths, exact specification of functions and interfaces of the systems' parts are necessary, as well as specification of their communication and deployment. Therefore, the information systems of organisations are realised as networks of quite autonomous, but cooperative, units communicating asynchronously via messages of appropriate format [7]. Unfortunately, design and implementation of those systems have to deal with many problems including the ability to clone components and to move the copies across a network (i.e. *component mobility*), creation, destruction and updating of components and connections during the systems' run-time (i.e. *dynamic reconfiguration*), maintaining components' compatibility, etc. [6]

Moreover, distributed information systems are getting involved. Their architectures are evolving during a run-time and formal specifications are necessary, particularly in critical applications. Design of the systems with *dynamic architectures* (i.e. architectures with dynamic reconfigurations) and *mobile architectures* (i.e. dynamic architectures with component mobility) can not be done by means of conventional software design methods. In most cases, these methods are able to describe semi-formally only sequential processing or simple concurrent processing bounded to one com-

ponent without advanced features such as dynamic reconfiguration.

The *component-based development* (CBD, see [17]) is a software development methodology, which is strongly oriented to composability and re-usability in a software system's architecture. In the CBD, from a structural point of view, a software system is composed of *components*, which are self contained entities accessible through well-defined *interfaces*. A connection of compatible interfaces of cooperating components is realised via their *bindings* (connectors). Actual organisation of interconnected components is called *configuration*. *Component models* are specific meta-models of software architectures supporting the CBD, which define syntax, semantics and composition of components.

Although the CBD can be the right way to cope with the problems of the distributed information systems, it has some limitations in formal description, which restrict the full support for the mobile architectures. Those restrictions can be delimited by usage of formal bases that do not consider dynamic reconfigurations and component mobility, strict isolation of control and business logic of components that does not allow full integration of dynamic reconfigurations into the components, etc.

This paper proposes a high-level component model addressing the mentioned issues. The model allows dynamic reconfigurations and component mobility, defined combination of control and business logic of components, and separation of a component's specification from its implementation. The paper also introduces a formal basis for description of the component model's semantics, i.e. the structure and behaviour of the components.

The remainder of this paper is organised as follows. In Section 2, we introduce the component model in more detail. In Section 3, we provide the formal basis for description of the component model. In Section 5, we review main approaches that are relevant to our subject. In Section 6, we discuss advantages and disadvantages of our component model and its formal description compared with the reviewed approaches and outline the future work. To conclude, in Section 7, we summarise our approach and current results.

2. Component Model

In this section, we describe our approach to the component model. The component model is presented in two views: structural and behavioural. At first, in Section 2.1, we introduce the component model's meta-model, which describes basic entities of the component model and their relations and properties. The second view, in Section 2.2, is focused on behaviour of the component model's entities, especially on the component mobility.

2.1. Meta-model

The Figure 1 describes an outline of the component model's meta-model[1] in the UML notation [20]. Three basic entities represent the core entities of a component based architecture: a component, an interface and a binding (a connector).

The *component* is an active communicating entity in a component based software system. In our approach, the component consists of component abstraction and component implementation. The *component abstraction* (`CompAbstraction` in the meta-model) represents the component's specification and behaviour given by the component's formal description (semantics of services provided by the component). The *component implementation* (`CompImplementation`) represents a specific implementation of the component's behaviour (an implementation of the services). The implementation can be primitive or composite. The *primitive implementation* (`CompImplPrimitive`) is realised directly, beyond the scope of architecture description (it is "a black-box"). The

[1] The figured diagram can not describe additional constraints, e.g. a composite component "contains" bindings that interconnect only interfaces of the component's subcomponents, not interfaces of its neighbouring components, etc.

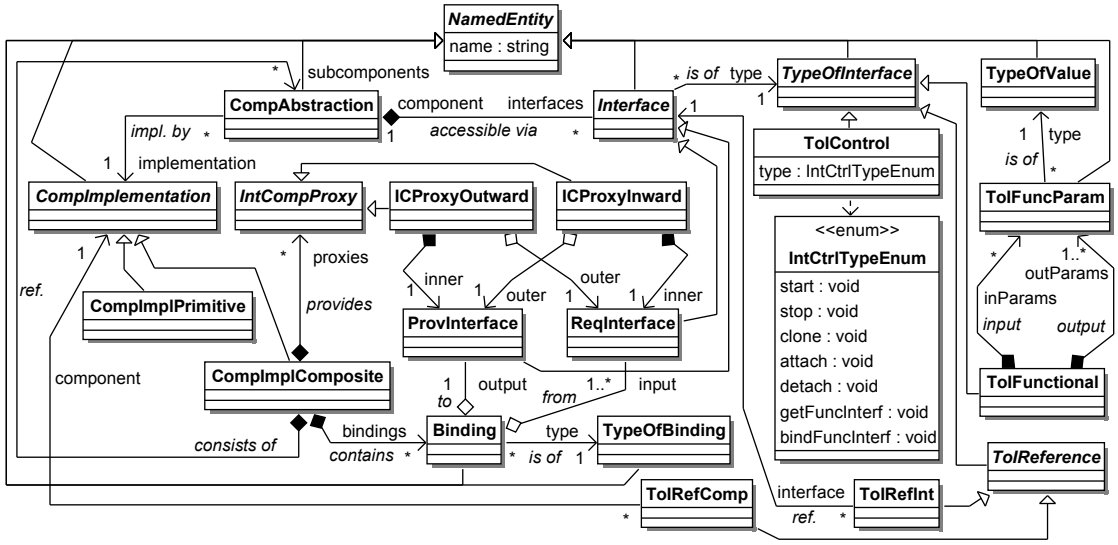


Figure 1. The meta-model of the component model (the UML notation [20])

composite implementation (`CompImplComposite`) is decomposable on a system of subcomponents at the lower level of architecture description (it is "a grey-box"). Those subcomponents are represented by component abstractions (`CompAbstraction` and relation "consists of").

Interfaces of a component are described in relation to the component's abstraction (relation "accessible via" from `CompAbstraction`). We distinguish two types of *interfaces*: required and provided (`ReqInterface` and `ProvInterface`, respectively), according to the type of services required or provided by the component from or to its neighbouring components, respectively, at the same level of hierarchy of components (i.e. not from or to subcomponents of a neighbouring component, for example). Moreover, the composite components' implementations (`CompImplComposite`) provide special *internal interfaces*, which are available only for the component's subcomponents and make accessible the component's external interfaces (i.e. the interfaces described in relation to `CompAbstraction`). The entity `ICProxyInward` connects a composite component's external provided interface to the component's internal required interface, while the entity `ICProxyOutward` connects a composite component's internal provided interface to the component's external required interface (the relations "outer" and "inner" and vice versa).

According to the functionality of interfaces, we can distinguish functional, control and reference interfaces (described by `TypeOfInterface`). The *functional interfaces* (`ToIFunct-ional`) represent business oriented services with typed input and output parameters (`ToIFuncParam` and `TypeOfValue`). The *control interfaces* (`ToIControl` and its attribute's `type`) provide services for obtaining references to a component's provided functional interfaces (type `getFuncInterfaces`), for binding a component's required functional interfaces (type `bind-FuncInterface`), and for changes of behaviour (types `start` and `stop`) and architecture. The services for changes of architecture are `clone`, `attach` and `detach` for obtaining references to a fresh copy of a component (type "cloning"), attaching of a new component as a subcomponent and detaching of an old subcomponent, respectively. The *reference interfaces* (`ToIReference`) are able to transmit references to components or interfaces, which is required to support component mobility.

Finally, the *binding* describes connection of required and provided interfaces of the identical types and of components at the same level of the hierarchy into a reliable communication link (entity `Binding`). The type of a binding (`TypeOfBinding`) can specify a communication

style (buffered and unbuffered connection), a type of synchronisation (blocking and output non-blocking), etc.

2.2. Behaviour and Support of Mobile Architectures

The previous section introduces the structure of the component model. A system described by means of the component model is one component with provided and required interfaces, which represent the system's input and output actions, respectively. The component can be implemented as a primitive component or as a composite component. The primitive component is realised directly, beyond the scope of architecture description, while the composite component is decomposable at the lower level of hierarchy into a system of subcomponents communicating via their interfaces and their bindings.

Behaviour of a primitive component has to be defined by a developer, simultaneously with definitions of the component's interfaces. The primitive component is defined as "a black-box", i.e. its behaviour can be described as a dependence relation of input and output actions. Behaviour of a composite component depends on behaviour of its subcomponents, but it includes also a description of communication between connected interfaces of those subcomponents and processing of specific control actions in the component (e.g. requests for starting or stopping of the component and their distribution to the component's subcomponents, etc.).

In the following description, we focus on the behaviour of control parts of components particularly related to *the features of mobile architectures*, i.e. on creation and destruction of components and connections and on passing of components. Evolution of a system's architecture begins in the state where its initialisation is finished.

A new component can be created as a copy of an existing component by means of its control interface `clone`. The resulting new component is deactivated (i.e. stopped) and packed into a message, which can be sent via outgoing connections into different location (via interfaces of type `ToIRefComp`) where it can be placed as a subcomponent of a parent component (by means of `attach` interface), connected to local neighbouring components (by means of `bindFuncInterf` and `getFuncInterf` interfaces) and activated (by means of `start` interface). *Destruction of an old component* can be done automatically after deactivating of the component (by means of `stop` interface), releasing of all its provided interfaces and disconnecting from its parent component (by means of `detach` interface).

Creation of new connections between two compatible functional interfaces can be done by means of passing of functional interfaces (via interfaces of type `ToIRefInt`). At first, a reference to provided functional interface (a target interface) is obtained from a component (via control interface `getFuncInterf`). This reference is sent via outgoing connections into different location (via interfaces of type `ToIRefInt`), but only in the same parent component and at the same level of hierarchy of components (i.e. crossing the boundary of a composite component is not allowed). The reference is received by a component with compatible required functional interface (a source interface) and a binding of this interface to referenced interface is created (by means of control interface `bindFuncInterf`). *Destruction of a connection* can be done by rebinding of a required interface participating in this connection.

As it follows from the description of behaviour, the connections can interconnect only interfaces of the same types. Moreover, dynamic creation of new connections and destruction of existing connection are permitted only for functional interfaces (type `ToIFunctional`). Those *restrictions*, together with the restriction of passing of interfaces' references described in the previous paragraph, prevent *architectural erosion* and *architectural drift* [11], which are caused by uncontrollable evolution of dynamic and mobile architecture resulting into degradation of the components' dependencies over time. In the component model, the architecture of control interfaces and their interconnections, which allow evolution and component mobility, is a static architecture.

Despite those restrictions, combining of actions of functional interfaces with actions of control interfaces is permitted inside primitive components. This allows to build systems where functional (business) requirements imply changes of a systems' architectures.

3. Formal Description

In this section, formal description of behaviour of the component model's entities is presented. The Section 3.1 provides an introduction to the process algebra π-calculus, which is used in description in Section 3.2. The description is based on our previous research on distributed information systems as systems of asynchronous concurrent processes [13] and the mobile architecture's features in such systems [15, 14].

3.1. The π-Calculus

The process algebra π-*calculus*, known also as a *calculus of mobile processes* [10], is an extension of Robin Milner's *calculus of communicating systems* (CCS). This section briefly summarises the fundamentals of the π-calculus, a theory of mobile processes, according to [16]. The following theoretical background is required for the component model's formal description in Section 3.2. The π-calculus allows modelling of systems with dynamic communication structures (i.e. mobile processes) by means of two concepts:

a process – an active communicating entity in a system, primitive or expressed in π-calculus (denoted by uppercase letters in expressions)[2],

a name – anything else, e.g. a communication link (a port), variable, constant (data), etc. (denoted by lowercase letters in expressions)[3].

Processes use names (as communication links) to interact, and pass names (as variables, constants, and communication links) to another process by mentioning them in interactions. The names received by a process can be used and mentioned by it in further interactions (as communication links). This "passing of names" permits mobility of communication links.

Processes evolve by performing actions. The capabilities for action are expressed via three kinds of prefixes ("output", "input" and "unobservable", as it is described later). We can define the π-calculus processes, their subclass and the prefixes as follows.

Definition 1 (π-calculus). *The* processes*, the* summations*, and the* prefixes *of the π-calculus are given respectively by*

$$\begin{aligned} P &::= M \mid P \mid P' \mid (z)P \mid !P \\ M &::= 0 \mid \pi.P \mid M + M' \\ \pi &::= \overline{x}\langle y\rangle \mid x(z) \mid \tau \end{aligned}$$

We give a brief, informal account of semantics of π-calculus processes. At first, process 0 is a π-calculus process that can do nothing, it is the *null process* or *inaction*. If processes P and P' are π-calculus processes, then following expressions are also π-calculus processes with formal syntax according to the Definition 1 and given informal semantics:

- $\overline{x}\langle y\rangle.P$ is an *output prefix* that can send name y via name x (i.e. via the communication link x) and continue[4] as process P,
- $x(z).P$ is an *input prefix* that can receive any name via name x and continue as process P with the received name substituted for every free occurrence[5] of name z in the process,
- $\tau.P$ is an *unobservable prefix* that can evolve invisibly to process P, it can do an internal (silent) action and continue as process P,
- $P + P'$ is a *sum* of capabilities of P together with capabilities of P' processes, it proceeds as either process P or process P', i.e. when a sum exercises one of its capabilities, the others are rendered void,
- $P \mid P'$ is a *composition* of processes P and P', which can proceed independently and can interact via shared names,

[2] A parametric process is also called "an agent".
[3] The names can be called according to their meanings (e.g. a port/link, a message, etc.).
[4] The prefix ensures that process P can not proceed until a capability of the prefix has been exercised.
[5] See the Definition 2.

- $(z)P$ is a *restriction* of the scope[6] of name z in process P,
- $!P$ is a *replication* that means an infinite composition of processes P or, equivalently, a process satisfying the equation $!P = P \mid !P$.

The π-calculus has two name-binding operators. The binding is defined as follows.

Definition 2 (Binding). *In each of $x(z).P$ and $(z)P$, the displayed occurrence of z is* binding *with* scope P. *An occurrence of a name in a process is* bound *if it is, or it lies within the scope of, a binding occurrence of the name, otherwise the occurrence is* free.

In our notations, we will omit a transmitted name, the second parts of input and output prefixes in a π-calculus expression, if it is not used anywhere else in its scope (e.g. instead of $(x)((y)\overline{x}\langle y\rangle.0 \mid x(z).0)$, we can write $(x)(\overline{x}.0 \mid x.0)$).

Since the sum and composition operators are associative and commutative (according to the relation of structural congruence [10]) they can be used with multiple arguments, independently of their order. Also an order of application of the restriction operator is insignificant. We will use the following notations:

- for $m \geq 3$, let $\prod_{i=1}^{m} P_i = P_1 \mid P_2 \mid \ldots \mid P_m$ be a *multi-composition* of processes $P_1, \ldots, P_m$, which can proceed independently and can interact via shared names,
- for $n \geq 2$ and $\tilde{x} = (x_1, \ldots, x_n)$, let $(x_1)(x_2)\ldots(x_n)P = (x_1, x_2, \ldots, x_n)P = (\tilde{x})P$ be a *multi-restriction* of the scope of names $x_1, \ldots, x_n$ to process P.

We will omit the null process if the meaning of the expression is unambiguous according to the above-mentioned equations (e.g. instead of $\overline{x}\langle y\rangle.0 \mid x(z).0$, we can write $\overline{x}\langle y\rangle \mid x(z)$). Moreover, the following equations are true for the *null process*:

$$M + 0 = M \qquad P \mid 0 = P \qquad (x)0 = 0$$

The π-calculus processes can be parametrised. A parametrised process, an abstraction, is an expression of the form $(x).P$. We may also regard abstractions as components of input-prefixed processes, viewing $a(x).P$ as an abstraction located at name a. In $(x).P$ as in $a(x).P$, the displayed occurrence of x is binding with scope P.

Definition 3 (Abstraction). *An* abstraction *of* arity $n \geq 0$ *is an expression of the form $(x_1, \ldots, x_n).P$, where the x_i are distinct. For $n = 1$, the abstraction is a monoadic abstraction, otherwise it is a polyadic abstraction.*

When an abstraction $(x).P$ is applied to an argument y it yields process $P\{y/x\}$. Application is the destructor of abstractions. We can define two types of application: pseudo-application and constant application. The pseudo-application is defined as follows.

Definition 4 (Pseudo-application). *If $F \stackrel{def}{=} (\tilde{x}).P$ is of arity n and $\tilde{y}$ is length n, then $P\{\tilde{y}/\tilde{x}\}$ is an* instance *of F. We abbreviate $P\{\tilde{y}/\tilde{x}\}$ to $F\langle\tilde{y}\rangle$. We refer to this instance operation as* pseudo-application *of an abstraction.*

In contract to the pseudo-application that is only abbreviation of a substitution, the constant application is a real syntactic construct. It allows to describe a recursively defined process.

Definition 5 (Constant application). *A* recursive definition *of a process constant K is an expression of the form $K \stackrel{\Delta}{=} (\tilde{x}).P$, where $\tilde{x}$ contains all names that have a free occurrence in P. A* constant application*, sometimes referred as an* instance *of the process constant K, is a form of process $K\lfloor\tilde{a}\rfloor$.*

Communication between processes (a computation step) is formally defined as a *reduction relation* $\rightarrow$. It is the least relation closed under a set of reduction rules.

Definition 6 (Reduction). *The* reduction relation*, $\rightarrow$, is defined by the following rules:*

$$\text{R-Inter}\ \frac{}{(\overline{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \rightarrow P_1 \mid P_2\{y/z\}}$$

$$\text{R-Tau}\ \frac{}{\tau.P + M \rightarrow P}$$

$$\text{R-Par}\ \frac{P_1 \rightarrow P_1'}{P_1 \mid P_2 \rightarrow P_1' \mid P_2}$$

$$\text{R-Res}\ \frac{P \rightarrow P'}{(z)P \rightarrow (z)P'}$$

$$\text{R-Struct}\ \frac{P_1 = P_2 \rightarrow P_2' = P_1'}{P_1 \rightarrow P_1'}$$

$$\text{R-Const}\ \frac{}{K\lfloor\tilde{a}\rfloor \rightarrow P\{\tilde{a}/\tilde{x}\}}\ K \stackrel{\Delta}{=} (\tilde{x}).P$$

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

The communication is described by the main reduction rule R-Inter. It means that a composition of a process proceeding as either process M_1 or the process, which sends name y via name x and continues as process P_1, and a process proceeding as either process M_2 or the process, which receives name z via name x and continues as process P_2, can perform a reduction step. After this reduction, the process is $P_1 \mid P_2\{y/z\}$ (all free occurrences of z in P_2 are replaced by y).

3.2. Description of the Component Model

A *software system* can be described by means of the component model as one component with provided and required interfaces, which represent the system's input and output actions, respectively. According to the component model's definition, every component can be implemented as a primitive component or as a composite component. Since a *primitive component* is realised as "a black-box", its behaviour has to be defined by its developer. This behaviour can be formally described as a π-calculus process, which uses names representing the component's interfaces, but also implements specific control actions provided by the component (e.g. requests to start or stop the component). On the contrary, a *composite component* is decomposable at the lower level of hierarchy into a system of subcomponents communicating via their interfaces and their bindings (the component is "a grey-box"). Formal description of the composite component's behaviour is a π-calculus process, which is composition of processes representing behaviour of the component's subcomponents, processes implementing communication between interconnected interfaces of the subcomponents and internal interfaces of the component and processes realising specific control actions (e.g. the requests to start or stop the composite component, but including their distribution to the component's subcomponents, etc.).

Before we define π-calculus processes implementing the behaviour of a component's individual parts, we need to define the *component's interfaces* within the terms of the π-calculus, i.e. as names used by the processes. The following names can be used in external or internal view of a component, i.e. for the component's neighbours or the composite component's subcomponents, respectively.

- external: $s_0, s_1, c, r_1^s, \ldots, r_n^s, p_1^g, \ldots, p_m^g$ (of a primitive or composite component),
- internal: $a, r_1'^s, \ldots, r_m'^s, p_1'^g, \ldots, p_n'^g$ (of a composite component only),

where n is a number of the component's required functional interfaces, m is a number of the component's provided functional interfaces (both from the external view) and the names have the following semantics:

via s_0 – a running component accepts a request for its stopping, which a composite component distributes also to all its subcomponents,

via s_1 – a stopped component accepts a request for its starting, which a composite component distributes also to all its subcomponents,

via c – a component accepts a request for its cloning and returns a new stopped instance of the component as a reply,

via r_i^s – a component accepts a request for binding given provided functional interface (included in the request) to the required functional interface r_i,

via p_j^g – a component accepts a request for referencing to the provided functional interface p_j that is returned as a reply,

via a – a composite component accepts a request for attaching its new subcomponent, i.e. for attaching the subcomponent's s_0 and s_1 names (stop and start interfaces), which can be called when the composite component will be stopped or started, respectively, and as a reply, it returns a name accepting the request to detach the subcomponent.

We should remark that there is a relationship between the names representing functional interfaces in the external view and the names representing corresponding functional interfaces in the internal view of the composite component. The composite component connects its external functional interfaces $r_1, \ldots, r_n$ (required) and $p_1, \ldots, p_m$ (provided) accessible via names

$r^s_1, \ldots, r^s_n$ and $p^g_1, \ldots, p^g_m$, respectively, to internal functional interfaces $p'_1, \ldots, p'_n$ (provided) and $r'_1, \ldots, r'_m$ (required) accessible via names $p'^g_1, \ldots, p'^g_n$ and $r'^s_1, \ldots, r'^s_m$, respectively. Requests received via external functional provided interface p_j are forwarded to the interface, which is bound to internal functional required interface r'_j (and analogously for interfaces p'_i and r_i).

3.2.1. Interface's References and Binding

At first, we define an auxiliary process $Wire$[7], which can receive a message via name x (i.e. input) and send it to name y (i.e. output) repeatedly till the process receives a message via name d (i.e. disable processing).

$$Wire \triangleq (x,y,d).(x(m).\overline{y}\langle m\rangle.Wire\lfloor x,y,d\rfloor \ + \ d)$$

Binding of a component's functional interfaces is done via control interfaces. These control interfaces provide references to a component's functional provided interfaces and allow to bind a component's functional required interfaces to referenced fictional provided interfaces of another local components. Process $Ctrl_{Ifs}$ implementing the control interfaces can be defined as follows

$$\begin{aligned}
SetIf &\triangleq (r,s,d).s(p).(\overline{d}.Wire\lfloor r,p,d\rfloor \mid SetIf\lfloor r,s,d\rfloor)\\
GetIf &\stackrel{def}{=} (p,g).g(r).\overline{r}\langle p\rangle\\
Plug &\stackrel{def}{=} (d).d\\
Ctrl_{Ifs} &\stackrel{def}{=} (r_1,\ldots,r_n,r^s_1,\ldots,r^s_n,p_1,\ldots,p_m,p^g_1,\ldots,p^g_m)\\
&\quad .(\prod_{i=1}^{n}(r^d_i)(Plug\langle r^d_i\rangle \mid SetIf\lfloor r_i,r^s_i,r^d_i\rfloor) \mid \prod_{j=1}^{m}!GetIf\langle p_j,p^g_j\rangle)
\end{aligned}$$

where names $r_1, \ldots, r_n$, $r^s_1, \ldots, r^s_n$, $p_1, \ldots, p_m$, $p^g_1, \ldots, p^g_m$ have been defined at the beginning of Section 3.2. Let us assume $Ctrl_{Ifs}$ shares its names $r_1, \ldots, r_n$ and $p_1, \ldots, p_m$ with a process implementing a component's core functionality via its required and provided interfaces, respectively. Pseudo-application $GetIf\langle p_j, p^g_j\rangle$ enables process $Ctrl_{Ifs}$ to receive a name x via p^g_j and to send p_j via name x as a reply (it provides a reference to an interface represented by p_j). Constant application $SetIf\lfloor r_i, r^s_i, r^d_i\rfloor$ enables process $Ctrl_{Ifs}$ to receive a name x via r^s_i, which will be connected to r_i by means of a new instance of process $Wire$ (it binds a required interface represented by r_i to a provided interface represented by x). To remove a former connection of r_i, a request is sent via r^d_i (in case it is the first connection of r_i, i.e. there is no former connection, the request is accepted by pseudo-application $Plug\langle r^d_i\rangle$).

In a composite component, the names representing external functional interfaces $r_1, \ldots, r_n$, $p_1, \ldots, p_m$ are connected to the names representing internal functional interfaces $p'_1, \ldots, p'_n$, $r'_1, \ldots, r'_m$. Requests received via external functional provided interface p_j are forwarded to the interface, which is bound to internal functional required interface r'_j (and analogously for interfaces p'_i and r_i). This is described in process $Ctrl_{EI}$.

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

[7] The process will be used also in the following parts of Section 3.2.

3.2.2. Control of a Component's Life-cycle

Control of a composite component's life-cycle[8] can be described as process $Ctrl_{SS}$.

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

where names s_0 and s_1 represent the component's interfaces that accept stop and start requests, respectively, and name a that can be used to attach a new subcomponent's stop and start interfaces (at one step).

The requests for stopping and starting the component are distributed to its subcomponents via names p_0 and p_1. Constant application $Life\lfloor s_1,s_0,p_1,p_0\rfloor$ enables process $Ctrl_{SS}$ to receive a message m via s_0 or s_1. Message m is distributed to the subcomponents by means of constant application $Dist\lfloor p_x,m,r\rfloor$ via shared name p_x, which can be p_0 in case the component is running or p_1 in case the component is stopped. When all subcomponents accepted message m, it is announced via name r and the component is running or stopped and ready to receive a new request to stop or start, respectively.

Pseudo-application $Attach\langle a,p_0,p_1\rangle$ enables process $Ctrl_{SS}$ to receive a message via a, a request to attach a new subcomponent's stop and start interfaces represented by names c_0 and c_1, respectively. The names are connected to p_0 and p_1 via new instances of processes $Wire$. Third name received via a, c_d, can be used later to detach the subcomponent's previously attached stop and start interfaces.

3.2.3. Cloning of Components and Updating of Subcomponents

Cloning of a component allows to transport the component's fresh copy into different location, i.e. its subsequent attaching as a subcomponent of other component. The processes of the cloning can be described as follows

$$
\begin{aligned}
Ctrl_{clone} \triangleq\; & (x).x(k).(s_0,s_1,c,r_1^s,\ldots,r_n^s,p_1^g,\ldots,p_m^g,r,p)\\
& (\overline{k}\langle s_0,s_1,c,r,p\rangle \mid \overline{r}\langle r_1^s,\ldots,r_n^s\rangle \mid \overline{p}\langle p_1^g,\ldots,p_m^g\rangle\\
& \mid Component\langle s_0,s_1,c,r_1^s,\ldots,r_n^s,p_1^g,\ldots,p_m^g\rangle \mid Ctrl_{clone}\lfloor x\rfloor)
\end{aligned}
$$

where pseudo-application $Component\langle s_0,s_1,c,\, r_1^s,\ldots,r_n^s,p_1^g,\ldots,p_m^g\rangle$ with well-defined parameters describes behaviour of the cloned component. When process $Ctrl_{clone}$ receives a request k via name x, it sends names s_0,s_1,c,r,p via name k as a reply. The first three names represent "stop", "start" and "clone" interfaces of a fresh copy of the component. The process is also ready to send names representing functional requested and provided interfaces of the new component, i.e. names $r_1^s,\ldots,r_n^s$ via name r names $p_1^g,\ldots,p_m^g$ via name p, respectively, and to receive a new request.

The fresh copy of a component can be used to replace a compatible subcomponent of a composite component. The process of update, which describes the replacing of an old subcomponent with a new one, is not mandatory part of the composite component's behaviour and its implementation

[8] A primitive component handles stop and start interfaces directly.

depends on particular configuration of the component (e.g. if the component allows updating of its subcomponents, a context of the replaced subcomponent, which parts of the component have to be stopped during the updating, etc.). As an illustrative case, we can describe process $Update$ as follows

$$\begin{aligned} Update \triangleq\ & (u, a, s_0, s_d, r^s_1, \ldots, r^s_m, p^g_1, \ldots, p^g_n)(k, s'_d) \\ & .(\overline{u}\langle k\rangle.k(s'_0, s'_1, c, r', p').\overline{s_0}.\overline{a}\langle s'_0, s'_1, s'_d\rangle.\overline{s_d} \\ & .r'(r'^s_1, \ldots, r'^s_n).(x)(\overline{p^g_1}\langle x\rangle.x(p).\overline{r'^s_1}\langle p\rangle \ldots \overline{p^g_n}\langle x\rangle.x(p).\overline{r'^s_n}\langle p\rangle) \\ & .p'(p'^g_1, \ldots, p'^g_m).(x)(\overline{p'^g_1}\langle x\rangle.x(p).\overline{r^s_1}\langle p\rangle \ldots \overline{p'^g_n}\langle x\rangle.x(p).\overline{r^s_m}\langle p\rangle) \\ & .\overline{s'_1}.Update\lfloor u, a, s'_0, s'_d, r^s_1, \ldots, r^s_m, p^g_1, \ldots, p^g_n\rfloor) \end{aligned}$$

Process $Update$ sends via name u a request for a fresh copy of a cloned component. As a return value, it receives a vector of names representing all functional interfaces in a process describing behaviour of the new component, which will replace an old subcomponent in its parent component implementing the update process. Name a provides the parent component's internal control interface to attach the new subcomponent's stop and start interfaces (the s'_0 and s'_1 names) and an interface later used to detach the subcomponent (name s'_d). Name s_0 is used to stop the replaced subcomponent and name s_d is needed to detach the old subcomponent's stop and start interfaces. Finally, names $r^s_1, \ldots, r^s_m$, $p^g_1, \ldots, p^g_n$ represent a context of the updated subcomponent, i.e. connected interfaces of neighbouring subcomponents.

3.2.4. Primitive and Composite Components

In conclusion, we can describe the complete behaviour of primitive and composite components. Let's assume that process abstraction $Comp_{impl}$ with parameters $s_0, s_1,\ r_1, \ldots, r_n,\ p_1, \ldots, p_m$ describes behaviour of the core of a primitive component (i.e. excluding processing of control actions), as it is defined by the component's developer. Further, let's assume that process abstraction $Comp_{subcomps}$ with parameters $a,\ r'^s_1, \ldots, r'^s_m,\ p'^g_1, \ldots, p'^g_n$ describes behaviour of a system of subcomponents interconnected by means of their interfaces into a composite component (see Section 3.2.1). Names $s_0, s_1,\ r_1, \ldots, r_n,\ p_1, \ldots, p_m$ and names $a,\ r^s_1, \ldots, r^s_m,\ p^g_1, \ldots, p^g_n$ are defined at the beginning of Section 3.2.

Processes $Comp_{prim}$ and $Comp_{comp}$ representing behaviour of the mentioned primitive and composite components can be described as follows

$$\begin{aligned} Comp_{prim} \stackrel{def}{=}\ & (s_0, s_1, c, r^s_1, \ldots, r^s_n, p^g_1, \ldots, p^g_m)(r_1, \ldots, r_n, p_1, \ldots, p_m) \\ & .(Ctrl_{Ifs}\langle r_1, \ldots, r_n, r^s_1, \ldots, r^s_n, p_1, \ldots, p_m, p^g_1, \ldots, p^g_m\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Comp_{impl}\langle s_0, s_1, r_1, \ldots, r_n, p_1, \ldots, p_m\rangle) \end{aligned}$$

$$\begin{aligned} Comp_{comp} \stackrel{def}{=}\ & (s_0, s_1, c, r^s_1, \ldots, r^s_n, p^g_1, \ldots, p^g_m) \\ & .(a, r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n) \\ & (Ctrl_{Ifs}\langle r_1, \ldots, r_n, r^s_1, \ldots, r^s_n, p_1, \ldots, p_m, p^g_1, \ldots, p^g_m\rangle \\ & \mid Ctrl_{Ifs}\langle r'_1, \ldots, r'_m, r'^s_1, \ldots, r'^s_m, p'_1, \ldots, p'_n, p'^g_1, \ldots, p'^g_n\rangle \\ & \mid Ctrl_{EI}\langle r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Ctrl_{SS}\langle s_0, s_1, a\rangle \mid Comp_{subcomps}\langle a, r'^s_1, \ldots, r'^s_m, p'^g_1, \ldots, p'^g_n\rangle) \end{aligned}$$

where processes $Ctrl_{Ifs}$ represent behaviour of control parts of components related to their interfaces (see Section 3.2.1), processes $Ctrl_{clone}$ describe behaviour of a control part of components related to cloning of these components (see Section 3.2.3), process $Ctrl_{SS}$ represents behaviour of

a component's control part handling its stop and start requests (see Section 3.2.2), and process $Ctrl_{EI}$ describes behaviour of communication between internal and external functional interfaces of a component (see Section 3.2.1).

4. An Example

As an example, we describe a component based system for user authentication and access control. At first the system receives an input from an user in form $(username, password)$ and verifies the user's password in order to check the user's identity. If the user's password passes the verification, the system creates a new session handle reserved for the user. The session handle is connected to the system's core. It enables the user to access the system's core functionality and performs the access control according to the user's authorisation. Finally, the session handle is passed back to the user as a return value of the whole process.

The system is composed of

- LOGIN component verifying the user's authentication and initiating the new session,
- CORE component providing the system's core functionality,
- and SESSION component enabling the user to access the CORE component according to the user's authorisation.

For simplicity, let's assume that component SESSION has only one input interface for the user's calls of the system's core without any explicit authorisation checks and component CORE implements simple shared memory – one storage for all users with two interfaces: for saving and loading a value to and from the memory, respectively.

4.1. Definition of the Components' Implementations

At first, we describe behaviour of cores of primitive components, i.e. the components' implementations, which have to be defined by developer of the system (see Section 3.2.4). Description of behaviour of the CORE component's implementation is:

$$Core_{impl} \stackrel{def}{=} (s_0, s_1, p_{save}, p_{load})(val)Core'_{impl}\lfloor \texttt{undef}, p_{save}, p_{load}\rfloor$$

$$\begin{aligned} Core'_{impl} \stackrel{\Delta}{=} \; & (val, p_{save}, p_{load})(p_{save}(val').Core'_{impl}\lfloor val', p_{save}, p_{load}\rfloor \\ & + p_{load}(ret).(\overline{ret}\langle val\rangle \mid Core'_{impl}\lfloor val, p_{save}, p_{load}\rfloor) \end{aligned}$$

where process $Core_{impl}$ can save a message received via name p_{save} and load the saved message and send it as a reply on a request received via name p_{load}.

Description of behaviour of the SESSION component's implementation is the following:

$$Session_{impl} \stackrel{def}{=} (s_0, s_1, r_{save}, r_{load}, p_{handle})Session'_{impl}\langle r_{save}, r_{load}, p_{handle}\rangle$$

$$\begin{aligned} Session'_{impl} \stackrel{def}{=} \; & (r_{save}, r_{load}, p_{handle})(save, load)(p_{handle}(ret) \\ & .\overline{ret}\langle save, load\rangle.Session'_{impl}\lfloor r_{save}, r_{load}, p_{handle}, save, load\rfloor) \end{aligned}$$

$$\begin{aligned} Session''_{impl} \stackrel{\Delta}{=} \; & (r_{save}, r_{load}, p_{handle}, save, load) \\ & (save(call).\overline{r_{save}}\langle call\rangle.Session'_{impl}\langle r_{save}, r_{load}, p_{handle}\rangle \\ & + load(call).\overline{r_{load}}\langle call\rangle.Session'_{impl}\langle r_{save}, r_{load}, p_{handle}\rangle) \end{aligned}$$

where process $Session_{impl}$ can receive via name p_{handle} an user's request, which is specified subsequently by inputs via names *save* or *load*, and pass it to process $Core_{impl}$ via names r_{save} or r_{load} (the required interfaces), respectively.

Finally, behaviour of the LOGIN component's implementation can be defined as follows:

$$\begin{aligned} Login_{impl} \triangleq\ & (s_0, s_1, p_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}) \\ & p_{init}(username, password, ret) \\ & .(Login_{verify}\langle username, password, ok, fail\rangle \\ & \mid Login'_{impl}\lfloor sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}, ret, ok, fail\rfloor \\ & \mid Login_{impl}\lfloor s_0, s_1, p_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}\rfloor) \end{aligned}$$

$$\begin{aligned} Login'_{impl} \triangleq\ & (sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}, ret, ok, fail)(new, d', t) \\ & (fail.\overline{ret}\langle \texttt{error}\rangle \ +\ ok.\overline{session_{clone}}\langle new\rangle.new(s'_0, s'_1, clone', r', p') \\ & .\overline{sys_{attach}}\langle s'_0, s'_1, d'\rangle.r'(r'^s_{save}, r'^s_{load}).p'(p'^g_{handle}) \\ & .\overline{core^g_{save}}\langle t\rangle.t(save).\overline{r'^s_{save}}\langle save\rangle.\overline{core^g_{load}}\langle t\rangle.t(load).\overline{r'^s_{load}}\langle load\rangle \\ & .p'^g_{handle}(handle).(\overline{s'_1} \mid \overline{ret}\langle handle\rangle)) \end{aligned}$$

where process $Login_{impl}$ can receive an user's initial request via name p_{init} as a triple of names (*username*, *password*, *ret*) and after successful verification of the user's name and password, the process returns a new session's handle via name *ret*. Name sys_{attach} provides an interface to attach new subcomponents into the system (see Section 3.2.2), name $session_{clone}$ is connected to a provided interface for cloning of SESSION component (see Section 3.2.3), and names $core^g_{save}$ or $core^g_{load}$ are connected to provided control interfaces for getting references to interfaces *save* or *load* of component CORE (see Section 3.2.1), respectively. The definition contains pseudo-application of process abstraction $Login_{verify}\langle username, password, ok, fail\rangle$, which represents description of behaviour of user's authentication process (e.g. $Login_{verify} \stackrel{def}{=} (\ldots).\overline{ok}$ for authorising of all users).

4.2. Description of the Component Based System

Now, we can describe behaviour of individual components including their control parts, as well as behaviour and structure of a composite component, which represents the whole component based system. According to Section 3.2.4, behaviour of components CORE and SESSION can be described as follows:

$$\begin{aligned} Core \stackrel{def}{=}\ & (s_0, s_1, c, p^g_{save}, p^g_{load}).(p_{save}, p_{load}) \\ & (Ctrl_{Ifs}\langle p_{save}, p_{load}, p^g_{save}, p^g_{load}\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Core_{impl}\langle s_0, s_1, p_{save}, p_{load}\rangle) \end{aligned}$$

$$\begin{aligned} Session \stackrel{def}{=}\ & (s_0, s_1, c, r^s_{save}, r^s_{load}, p^g_{handle}).(r_{save}, r_{load}, p_{handle}) \\ & (Ctrl_{Ifs}\langle r_{save}, r_{load}, r^s_{save}, r^s_{load}, p_{handle}, p^g_{handle}\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Session_{impl}\langle s_0, s_1, r_{save}, r_{load}, p_{handle}\rangle) \end{aligned}$$

Behaviour of component LOGIN has to be described differently from the others, because it uses control interfaces sys_{attach}, $session_{clone}$, $core^g_{save}$, $core^g_{load}$, which can not be referenced (contrary to functional interfaces, see Section 2.2). This case can be compared with the description of *Update*

process in Section 3.2.3. The behaviour of component LOGIN can be described as follows:

$$\begin{aligned} Login \stackrel{def}{=} \; & (s_0, s_1, c, p^g_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}).(p_{init}) \\ & (Ctrl_{Ifs}\langle p_{init}, p^g_{init}\rangle \mid Ctrl_{clone}\lfloor c \rfloor \\ & \mid Login_{impl}\lfloor s_0, s_1, p_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}\rfloor) \end{aligned}$$

Finally, behaviour and structure of a composite component, which represents the whole component based system, can be described as follows:

$$\begin{aligned} System \stackrel{def}{=} \; & (s_0, s_1, c, p^g_{init})(a, p_{init}, r'_{init}, r'^s_{init}) \\ & .(Ctrl_{Ifs}\langle p_{init}, p^g_{init}\rangle \mid Ctrl_{Ifs}\langle r'_{init}, r'^s_{init}\rangle \mid Ctrl_{EI}\langle p_{init}, r'_{init}\rangle \\ & \mid Ctrl_{clone}\lfloor c \rfloor \mid Ctrl_{SS}\langle s_0, s_1, a\rangle \mid System'\langle a, r'^s_{init}\rangle) \end{aligned}$$

$$\begin{aligned} System' \stackrel{def}{=} \; & (sys_{attach}, r^s_{init}) \\ & (p^g_{init}, core^g_{save}, core^g_{load}, sess^s_{save}, sess^s_{load}, sess^g_{handle}, login_{clone}, core_{clone}) \\ & (sess_{clone}, s_0^{login}, s_1^{login}, d^{login}, s_0^{core}, s_1^{core}, d^{core}, s_0^{sess}, s_1^{sess}, d^{sess}) \\ & (Login\langle s_0^{login}, s_1^{login}, login_{clone}, p^g_{init}, sys_{attach}, sess_{clone}, core^g_{save}, core^g_{load}\rangle \\ & \mid Core\langle s_0^{core}, s_1^{core}, core_{clone}, core^g_{save}, core^g_{load}\rangle \\ & \mid Session\langle s_0^{sess}, s_1^{sess}, sess_{clone}, sess^s_{save}, sess^s_{load}, sess^g_{handle}\rangle \\ & \mid \overline{sys_{attach}}\langle s_0^{login}, s_1^{login}, d^{login}\rangle \mid \overline{sys_{attach}}\langle s_0^{core}, s_1^{core}, d^{core}\rangle \\ & \mid \overline{sys_{attach}}\langle s_0^{sess}, s_1^{sess}, d^{sess}\rangle \mid \overline{p^g_{init}}\langle t\rangle.t(init).\overline{r^s_{init}}\langle init\rangle) \end{aligned}$$

5. Related Work

There have been proposed several component models [8]. In this section, we focus on two contemporary component models supporting some features of dynamic architectures and formal descriptions.

5.1. Fractal

The *component model Fractal* [3] is a general component composition framework with support for dynamic architectures. A Fractal component is formed out of two parts: a controller and a content. The content of a composite component is composed of a finite number of nested components. Those subcomponents are controlled by the controller ("a membrane") of the enclosing component. A component can be shared as a subcomponent by several distinct components. A component with empty content is called a primitive component. Every component can interact with its environment via operations at external interfaces of the component's controller, while internal interfaces are accessible only from the component's subcomponents. The interfaces can be of two sorts: client (required) and server (provided). Besides, a functional interface requires or provides functionalities of a component, while a control interface is a server interface with operations for introspection of the component and to control its configuration. There are two types of directed connections between compatible interfaces of components: primitive bindings between a pair of components and composite bindings, which can interconnect several components via a connector.

Behaviour of Fractal components can be formally described by means of *parametrised networks of communicating automata* language [2]. Behaviour of each primitive component is modelled as a finite state *parametrised labelled transition system* (pLTS) – a labelled transition system with parametrised actions, a set of parameters of the system and variables for each state. Behaviour of a composed Fractal component is defined using a *parametrised synchronisation network* (pNet). It is a pLTS computed as a product of subcomponents' pLTSs and a transducer. The transducer is a pLTS, which synchronises actions of the corresponding LTSs of the subcomponents. When synchronisation of the actions occurs, the transducer changes its state, which means reconfiguration of the component's architecture. Also behaviour of a Fractal component's controller can be formally described by means of pLTS/pNet. The result is composition of pLTSs for binding and unbinding of each of the component's functional interfaces (one pLTS per one interface) and pLTS for starting and stopping the component.

5.2. SOFA and SOFA 2.0

In the *component model SOFA* [12], a part of *SOFA project (SOFtware Appliances)*, a software system is described as a hierarchical composition of primitive and composite components. A component is an instance of a template, which is described by its frame and architecture. The frame is a "black-box" specification view of the component defining its provided and required interfaces. Primitive components are directly implemented by described software system – they have a primitive architecture. The architecture of a composed component is a "grey-box" implementation view, which defines first level of nesting in the component. It describes direct subcomponents and their interconnections via interfaces. The connections of the interfaces can be realised via connectors, implicitly for simple connections or explicitly. Explicit connectors are described in a similar way as the components, by a frame and architecture. The connector frame is a set of roles, i.e. interfaces, which are compatible with interfaces of components. The connector architecture can be simple (for primitive connectors), i.e. directly implemented by described software system, or compound (for composite connectors), which contains instances of other connectors and components.

The SOFA uses a *component definition language* (CDL) [9] for specification of components and *behaviour protocols* (BPs) for formal description of their behaviours. The BPs [21] are regular-like expressions on the alphabet of event tokens representing emitting and accepting method calls. Behaviour of a component (its interface, frame and architecture) can be described by a BP (interface, frame and architecture protocol, respectively) as the set of all traces of event tokens generated by the BP. The architecture protocols can be generated automatically from architecture description by a *CDL compiler*. A *protocol conformance relation* ensures the architecture protocol generates only traces allowed by the frame protocol. From dynamic architectures, the SOFA allows only a dynamic update of components during a system's runtime. The update consists in change of implementation (i.e. an architecture) of the component by a new one. Compatibility of the implementations is guaranteed by the conformance relation of a protocol of the new architecture and the component's frame protocol.

Recently, the SOFA team is working on a new version of the component model. The *component model SOFA 2.0* [5] aims at removing several limitations of the original version of SOFA – mainly the lack of support of dynamic reconfigurations of an architecture, well-structured and extensible control parts of components, and multiple communication styles among components.

6. Discussion and Future Work

The component model proposed in this paper is able to handle mobile architectures, unlike the SOFA that supports only a subset of dynamic architectures (implementing the update operation) or the Fractal/Fractive, which does not support components mobility. As is described in

Section 3.2, the π-calculus provides fitting formalism for description of software systems based upon the component model.

The proposed semantics of the component model permits to combine control interfaces and functional interfaces inside individual primitive components where the control actions can be invoked by the functional actions, i.e. by a system's business logic represented by business oriented services. This allows to build systems where functional (business) requirements imply changes of the systems' architectures. Regardless, in some cases, this feature can lead to *architectural erosion* and *architectural drift* [11], i.e. unpredictable evolution of the system's architecture. For that reason, the component model forbids dynamic changes of connections between control interfaces, which reduces architecture variability to patterns predetermined at a design-time. Formal description of the components integrating the control and functional actions can be compared with the transducer in the Fractal/Fractive approach (see Section 5.1).

The next feature of the component model is partially independence of a component's specification from its implementation (see the description of entities `CompAbstraction` and `CompImplementation` in Section 2.1). This feature is similar to the SOFA's component-template relationship. It allows to control behaviour of a primary component's implementation, define a composite component's border that isolates its subcomponents, which is called "a membrane" in the Fractal, etc. (for comparison, see Section 5.1 and Section 5.2)

The attentive reader will have noticed that the process algebra π-calculus, as it is defined in Section 3.1 and applied to the formal description of behaviour of the component model's entities in Section 3.2, allows to describe only synchronous communication. Although, in most cases, we need to apply the component model to distributed software systems with asynchronous communication. This limitation is a consequence of the reduction relation's definition (see Definition 6 in Section 3.1). The problem can be solved by proposing of a "buffered" version of communication between interfaces (i.e. in process $Wire$ from Section 3.2.1) or, alternatively, by using of an asynchronous π-calculus [16].

The next important extension of the presented approach is application of typed π-calculus [10, 16], which allows to distinguish types of names. This feature is necessary to formally describe constraints of the type system of interfaces in behaviour of components. In the component model's metamodel, the type system is defined by instances of entity `TypOfInterface` and its descendants and related entities (see Section 2.1).

However, the above mentioned modifications are out of scope of this paper and a final version of the component model's formal description including the proposed extensions is part of current work. Further ongoing work is related to the realisation of a supporting environment, which allows integration of the component model into software development processes, including integration of verification tools and implementation support. The idea is to use results of the *ArchWare project* [1], especially for theorem-proving and model-checking[9]. We intend to use the *Eclipse Modeling Framework* (EMF) [4, 19] for modeling and code generation of tools based on the component model and the *Eclipse Graphical Modeling Framework* (GMF) [18] for developing graphical editors according to the rules described in the component model's metamodel (based on EMF).

7. Conclusion

In this paper, we have presented an approach, which contributes to specify component-based software systems with features of dynamic and mobile architectures. The proposed component model splits a software system into primitive and composite components according to decomposability of its parts, and the components' functional and control interfaces according to the types of required or provided services. The components can be described at different levels of abstraction, as their specifications and implementations.

[9] See the tools presented in documents D3.5b and D3.6c at [1].

Semantics of the component model's entities is formally described by means of the process algebra π-calculus (known as a calculus of mobile processes). Formal description of behaviour of a whole system can be derived from the visible behaviour of its primitive components and their compositions and communication, both defined at a design-time. The result is a π-calculus process, which describes the system's architecture, including its evolution and component mobility, and communication behaviour of the system. Thereafter, critical properties of the system can be verified by means of π-calculus model checker.

We are currently working on extending our approach to use asynchronous communication between components and a type system for their interfaces. Future work is related to integration of the component model into software development processes, including application of verification tools and implementation support. In the broader context, the research is a part of a project focused on formal specifications and prototyping of distributed information systems.

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

References

[1] ArchWare project. http://www.arch-ware.org/, Nov. 2006.

[2] T. Barros. *Formal specification and verification of distributed component systems.* PhD thesis, Université de Nice – INRIA Sophia Antipolis, Nov. 2005.

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

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework.* The Eclipse Series. Addison Wesley Professional, Aug. 2003.

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

[6] J. Král and M. Žemlička. Autonomous components. In *SOFSEM 2000: Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*. Springer, 2000.

[7] J. Král and M. Žemlička. Software confederations and alliances. In *CAiSE Short Paper Proceedings*, volume 74 of *CEUR Workshop Proceedings*, pages 229–232. CEUR-WS.org, 2003.

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

[9] V. Mencl. Component definition language. Master's thesis, Charles University, Prague, 1998.

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

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[12] F. Plášil, D. Bílek, and R. Janeček. SOFA/DCUP: Architecture for component trading and dynamic updating. In *4th International Conference on Configurable Distributed Systems*, pages 43–51, Los Alamitos, CA, USA, May 1998. IEEE Computer Society.

[13] M. Rychlý. Towards verification of systems of asynchronous concurrent processes. In *Proceedings of 9th International Conference Information Systems Implementation and Modelling (ISIM' 06)*, pages 123–130. MARQ, Apr. 2006.

[14] M. Rychlý. Component model with support of mobile architectures. In *Information Systems and Formal Models*, pages 55–62. Faculty of Philosophy and Science in Opava, Silesian University in Opava, Apr. 2007.

[15] M. Rychlý and J. Zendulka. Distributed information system as a system of asynchronous concurrent processes. In *MEMICS 2006 Second Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. Faculty of Information Technology BUT, 2006.

[16] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, First paperback edition, Oct. 2003.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, second edition, Nov. 2002.

[18] The Eclipse Foundation. Eclipse Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/, Sept. 2007.

[19] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/, Sept. 2007.

[20] Unified Modeling Language, version 1.5. Document formal/03-03-01, Object Management Group, 2003.

[21] S. Višňovský. *Modeling software components using behavior protocols*. PhD thesis, Dept. of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Bi-dimensional Composition with Domain Specific Languages

Anca Daniela Ionita*, Jacky Estublier**, Thomas Leveque**, Tam Nguyen**
* *University Politehnica of Bucharest, Automatic Control and Computers Faculty*
** *LIG-IMAG, Grenoble, France*
Anca.Ionita@mag.pub.ro, Jacky.Estublier@imag.fr, Thomas.Leveque@imag.fr, Tam.Nguyen@imag.fr

Abstract

The paper presents how domain modeling may leverage the hierarchical composition, supporting two orthogonal mechanisms (vertical and horizontal) for composing completely autonomous parts. The vertical mechanism is in charge of coordinating heterogeneous components, tools or services at a high level of abstraction, by hiding the technical details. The result of such a composition is called "domain" and represents a high granularity unit of reuse, which may be easily developed in Mélusine framework. A domain is characterised by a Domain Specific Language (DSL) and applications in that domain are defined by models executed by the DSL interpreter. Most often, this is significantly simpler than writing a program using a general purpose language. Unfortunately, DSLs have a narrow scope, while real world applications usually span over many domains, raising the issue of domain (and DSL) composition. To overcome this problem, the horizontal mechanism composes domains at the level of their DSLs, even if they have been independently designed and implemented. The paper presents a model and metamodel perspective of the Mélusine bi-dimensional composition, assisted and automated with the Codèle tool, which allows specification at a high level of abstraction, followed by Java and AspectJ code generation.

1. Introduction

In the widely adopted Component Based Software Engineering (CBSE) approach, components know each other, must have compatible interfaces and must comply with the constraints of the same component model, which reduces the likelihood of reusing components, and therefore the capability to obtain a large variety of assemblies. Therefore, alternative composition mechanisms have to be explored, such as to preserve the CBSE advantages (coming from hiding the internal structure and reusing components without any change) but to relax the rigidity of the composition constraints:

- The components or, generally speaking, the parts, should ignore each other, such that they could have been designed and developed independently, i.e. they do not call each other;
- Composed parts should be of any nature (ad hoc, legacy, COTS, local or distant);
- Parts should be allowed to be heterogeneous i.e. they do not need to follow a particular model (component model, service etc.);
- Parts should be reused without having to perform any change in their code.

The bi-dimensional composition mechanism presented here is intended to be a solution for such situations. The idea is to obtain composable elements that are not traditional components, but much larger units, called domains, which do not expose simple interfaces, but domain models, representing DSLs for specifying the application models.

One of the important problems to be solved was related to the heterogeneity of components, tools or services that have to be reused. A possible solution was to imagine that the part to be composed is wrapped into a "composable element" [22]. There was also a need to define a composition mechanism that is not based on the traditional method call, for composing parts that ignore each other, and therefore do not call each other. The publish/subscribe mechanism [2] was an interesting candidate, since the component that sends events ignores who (if any) is interested in that event, but the receiver knows and must declare what it is interested in. If other events, in other topics, are sent, the receiver code has to be changed. Moreover, the approach works fine only if the sender is an active component. A more appropriate solution for our requirements could be given by Aspect Oriented Software Development (AOSD) [18], [13], which eliminates some of the constraints above, since the sender (the main program) ignores and does not call the receiver (the aspects). Unfortunately, the aspect knows the internals of the main program, which defeats the encapsulation principle [8] and aspects are defined at a low level of abstraction (the code) [12], [24].

In our approach, heterogeneity is dealt with by coordinating components, tools or services from a higher abstraction level; this is what we call *vertical composition* and is attained by defining a domain DSL, which can natively specify entities specific to the domain and natively grasp the semantics (behaviour) of these entities within its interpreter; therefore, defining an application in the domain turns out to be the simple definition of a model in the DSL language. As usual, each domain is well instrumented with editors, interpreters, debuggers, analyzers, whose development is rather expensive, even with the help of the recent environments. Maybe more important, the practitioners acquire expertise in using these languages and benefit from a large set of existing models, which constitute a part of the company assets. Therefore, a large scale reuse of these domains is essential for the applicability of such an approach and is promoted through rich DSL semantics. Unfortunately, the richer the semantics embedded in the DSL, the simpler the models, but the narrower the language scope. In this context, the main drawback of DSLs comes out from the fact that most real life applications usually crosscut several domains, but they cannot be simply described by selecting a set of independent domain specific models, each one describing how the application behaves inside each covered domain.

Consequently there is also a need to compose domains; this is what we call *horizontal composition* and is not based on calling component interfaces, but on composing domain DSLs and models. In contrast to method call, model composition does not impose that models stick to common interfaces, or know each other, because one can either merge or relate independent concepts. Moreover, model composition allows the definition of variability points [17], which makes the mechanism more flexible than component composition.

For building applications spanning different domains, the challenge is to reuse the domain tools, the existing models and the practitioner's expertise and know-how; this is far from trivial and is not possible if one creates a new language for the composite domain. As discussed above, for obtaining a non-invasive method, a possibility is to adopt an implementation based on AOP (Aspect Oriented Programming); the composed domains and their models are totally unchanged and the new code is isolated with the help of aspects. However, since the AOP technique is at code level, performing domain composition has proved to be very difficult in practice; the conceptual complexity is increased, due to the necessity to deal with many technical details. This problem has been treated in many research works. The elevation of crosscutting modeling concerns to first-class constructs has been done in having [15], by generating weavers from domain specific descriptions, using ECL, an extension of OCL (Object Constraint Language). Another weaver constructed with domain modeling concepts is presented in [16], while [25] discusses mappings from a design-level language, Theme/UML, to an implementation-level language, AspectJ. Our solution is to clearly sep-

arate the specification of the composition from its implementation, by designing at a high conceptual level and then generating the code based on aspects.

For managing the complexity in a user friendly manner, the user defines the composition using wizards, for selecting among pre-defined properties. Designers and programmers are assisted by the Mélusine engineering environment for developing such autonomous domains, for composing them and for creating applications based on them [22]. For facilitating an easier domain composition, by generating Java and AspectJ code, Mélusine was leveraged by Codèle, a tool that guides the domain expert for performing the composition at the conceptual level, as opposed to the programming level.

Chapter 2 describes the architecture and the principles that stand behind the creation of domains driven by their DSLs and the composition at a high level of abstraction. Chapter 3 presents the metamodels that allow code generation for vertical and horizontal composition. Chapter 4 introduces some details related to the implementation choices, including some mappings for code generation. Chapter 5 compares the approach with other related works and evaluates its usefulness in respect with the domain compositions performed before the availability of the code generation facility offered by Codèle.

2. Bi-dimensional Composition Based on DSLs

The alternative composition idea presented above is to create units of reuse that are *autonomous* (eliminating dependencies on the context of use) and *composable* at an *abstract level* (eliminating dependencies on the implementation techniques and details). The solution presented here combines two techniques (see Fig. 1): *building autonomous domains* using *vertical composition* and *abstract composition of domains* using *horizontal composition*, performed between the abstract concepts of independent domains, without modifying their code.

2.1. Developing Autonomous Domains: Vertical Composition

Developing a domain can be performed following a top-down or a bottom-up approach. From a top down perspective, the required functionalities of the domain can be specified through a model, irrespective of its underlying technology. Then, one identifies the software artifacts (available or not) that will be used to implement the expected functionality and make them interoperate. From a bottom up perspective, the designer already knows the software artifacts that may be used for the implementation and will have to interoperate; therefore, the designer has to identify the abstract concepts shared by these software artifacts and how they are supposed to be consistently managed. Finally, one defines how to coordinate the software artifacts, based on the behavior of the shared concepts.

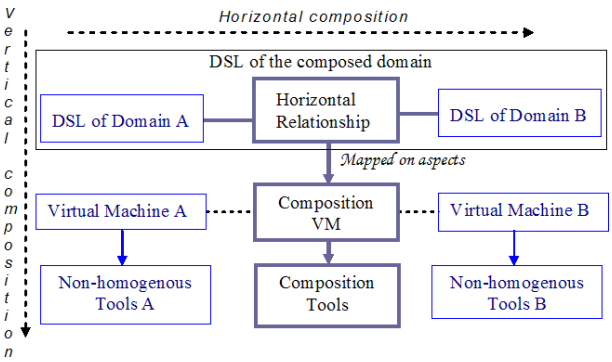


Figure 1. Bi-dimensional composition mechanism

In both cases, the composition is called vertical, because the real software components, services or tools are driven based on a high level model of the application. The model elements are instances of the shared concepts, which are abstractions of the actual software artifacts. The synchronization between these software artifacts and the model means that the evolution of the model is transformed into actions performed by the software artifacts.

The set of shared concepts and their consistency constraints constitute a domain model, to which the application model must conform to. In the Model Driven Engineering (MDE) vocabulary, the domain model is the metamodel,

or a DSL for all the application models for that domain [12].

The application models are interpreted by a virtual machine, built according to the domain DSL, which orchestrates the lower level services or tools. The domain interpreter is realized by Java classes that reify the shared concepts of the domain model and whose methods implement the behavior of these concepts. In many cases, these methods are empty, because most, if not all the behavior is actually delegated to other software artifacts, with the help of aspect technology. Thus, the domain interpreter, also called the domain virtual machine, separates the abstract and conceptual part from the implementation, creating 3 layers architecture [12]. The domains may be autonomously executed, they do not have dependencies and they may be easily used for developing applications.

2.1.1. Domain Specific Languages in Mélusine

The domain specific languages defined in Mélusine are rather small, covering a narrow domain and typically, they are object oriented. As usual, each language description contains two parts: syntax and semantics. The abstract syntax (AS) of the language contains the concepts and rules necessary to define a valid model, while its Semantic Domain (SD) is needed to provide the meaning of the abstract syntax concepts. By convention, the abstract syntax is defined by a class diagram, while the Semantic Domain is defined based on the methods pertaining to the AS classes, plus some additional classes. The concrete syntax (CS) is provided by a specific editor.

The *Product* domain, one of our intensely reused domains, is presented in the case study of this paper. It was developed as a basic versioning system for various products, characterised by a map of attributes, according to their type; the versions are stored in a tree, consisting of branches and revisions. The *Product* domain DSL is shown in Fig. 2 and contains both AS elements (light colored) and SD elements (dark colored).

From a Language Engineering point of view, this DSL is the definition of a language in which models are written; from a Domain Modeling point of view, it is a model of the application domain [12]. Thus, the DSL is the symbiosis of both views, since it is a language in which models are written, but, being Domain Specific, it contains the domain specific concepts, their allowed relationships and their behaviors. The DSL captures both the abstract syntax and the semantic aspects and it has different purposes: on one hand, it is used to develop models; on the other hand it is used to develop the interpreter and the editor of the domain and to compose domains for enlarging their scope. These activities involve an awareness of the concepts related to the semantic domain, which is necessary, for instance, for developing the interpreters, but also for composing them, in order to be able to compose domains.

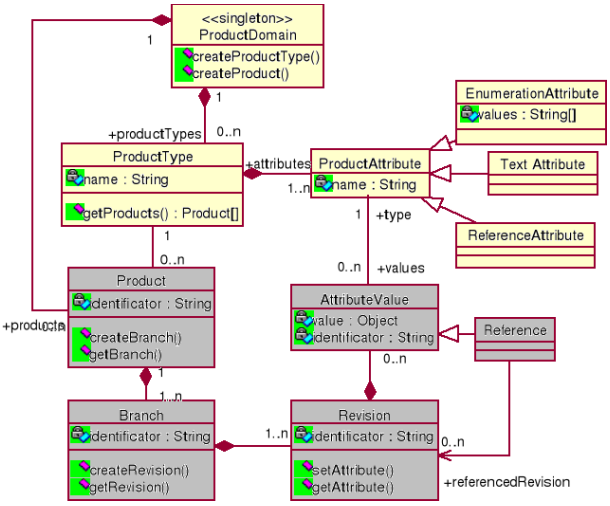


Figure 2. DSL of Product domain

2.1.2. Domain Specific Models

For defining an application, one creates a model that is going to be interpreted at run-time. Suppose we use our *Product* domain to version the software artefacts produced when developing an application based on the J2EE architecture. A *Servlet* in this application model conforms to the *ProductType* concept from the *Product* DSL.

In practice, the models can be expressed in several formalisms, and represented in a variety of ways; indeed, models may be defined in UML, or in Ecore, through generated editors (like in

most metamodeling environments), and stored in different formats, currently XML based. We have developed a number of filters, allowing one to define models and metamodels in these different formalisms, using different environments and editors. An example of editor for *Product* domain is given in Fig. 3. However, since our DSLs are written in Java, models always consist in a set of Java objects at execution. Models, expressed in various formalisms, will be transparently converted to Java objects at the beginning of interpretation phase. In most cases, when domains are narrow enough, the complete models semantics lies in the DSL. In this case, models are purely structural, and simple editors like those generated by EMF are sufficient. This is very important, because it allows non programmers to define executable models themselves. If models require specific semantics, it has to be described in Java.

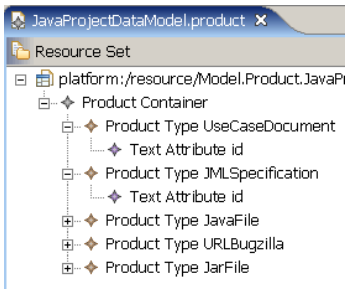


Figure 3. Model editor for Product domain

2.2. Abstract Domain Composition: Horizontal Composition

It may happen that the development of a new application requires the cooperation of two concepts, pertaining to two different domains, and realized through two or more software components, services or tools. In this case, the interoperation is performed through a horizontal composition between these abstract concepts, and also through the domain virtual machines, ignoring the low level components, services and tools used for the implementation. The mechanism consists in establishing relationships between concepts of the two DSLs and implementing them using aspect technology, such as to keep the composed domains unchanged. A very strict definition of the horizontal relationship properties is necessary, such as to be able to generate most of the AOP code for implementing them. This code belongs to the Composition Virtual Machine and is separated from the virtual machines of the composed domains.

This composition is called horizontal, because it is performed between parts situated at the same level of abstraction. It can be seen as a grey box approach, taking into account that the only visible part of a domain is its DSL. It is a non-invasive composition technique, because the components and adapters are hidden and are reused as they are. The composition result is a new domain model and therefore, a new domain, with its virtual machine, so that the process may be iterated. As the domains are executable and the composition is performed imperatively, its result is immediately executable, even if situated at a high level of abstraction.

Model composition is actually performed by creating links between model elements (instances of the DSL classes) so by instantiating the horizontal relationships defined at metamodel level. The choices of links ends may be made either automatically or manually (interactive) with the help of the application designer. Interactive selection is often used, since concepts of existing models may not match to each other perfectly (they may have different names, but the same meaning or have the same name, but behaviors that partially overlap) and no rule can be defined for it. However, it may be a tedious process, especially for composing large models. In contrast, automatic selection can relieve model designer from this burden and is particularly appreciated when models are very large. The default criterion for automatic selection can be based on name matching.

2.2.1. Horizontal Composition at Metamodel, Model and Execution Levels

A real example of domain composition, realized in our industrial applications, is illustrated in Fig. 4. On the left, the *Activity* domain supports workflow execution, while on the right, the *Prod-*

uct domain is meant to store typed products and their attributes. Each domain has a DSL (see the metamodel level). The upper part shows the visible concepts (the abstract syntax, in light grey) used for defining the models with appropriate editors; the lower part (in dark grey) shows the hidden classes, introduced for implementing the interpreters (the virtual machines) and for holding the state of the models during the execution process.

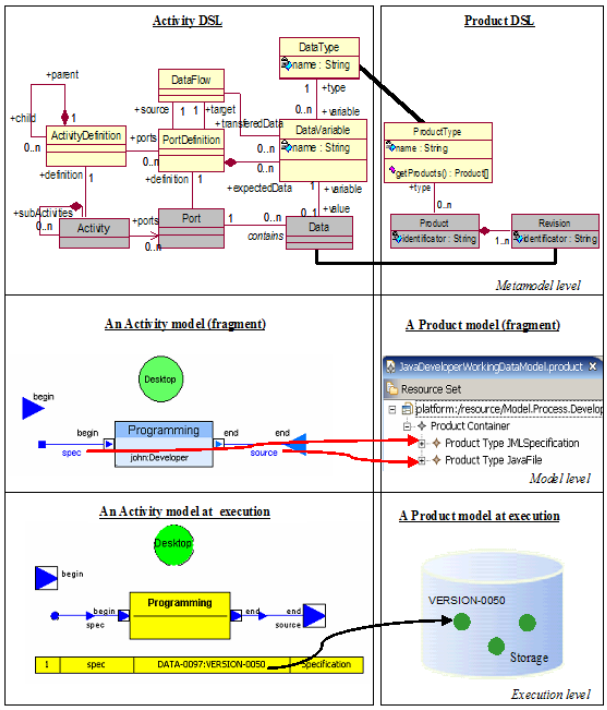


Figure 4. Composition of Activity and Product domains

For each domain, a model is made by instantiating the concepts found in the light colored part of the domain DSL. At model level, on the left, Figure 4 shows an Activity model, conforming to its DSL above. This model describes a very simple software development process, which only contains one activity – *Programming*; the box *Programming* is an instance of the *ActivityDefinition* concept. Labels on the activity connectors, like *spec* or *source* are instances of the *DataVariable* concept. These data variables correspond to instances of *DataType: Specification* and *Program* (not shown in this figure). Similarly, on the right side of Figure 4, at model level, there is a *Product* model, containing two instances of *ProductType: JMLSpecification* and *JavaFile*.

The *Activity* model in this example is made of a simple activity, in which a developer *john* receives a software specification *spec*, realizes the activity *Programming* and produces the source code *source*. However, the developer *john* may need to work on various revisions of his specification or of his source, so the *Activity* domain needs to be composed with the *Product* domain, for adding the versioning facility. These two domains (*Activity* and *Product*) are related together by horizontal relationships at metamodel level, for example, a horizontal relationship is defined between *DataType* and *ProductType* and another one between *Data* and *Revision*. At model level, a link relates the type of *spec* – *Specification* (found in the *Activity* model) – to *JMLSpecification*, instantiated from *ProductType* (found in the *Product* model). Another link relates *Program* (the type of *source*) to the *JavaFile* product type. These two links conform to the relationships defined between the *DataType* and *ProductType* concepts. At execution level, a data from the Activity model, for example DATA_0097 is related to a revision from Product model, for example *VERSION-0050* (see Fig. 4). This link conforms to the relationship between *Data* and *Revision*, situated at metamodel level.

Even if in the example above there was a clear correspondence between Specification from Activity model and JMLSpecification from Product Model, in practice, there may be several instances of a metamodel concept on both sides, as exemplified in Table 1. For creating the link at model level, one has to choose among these instances, such as to select a single one-to-one correspondence.

3. Metamodels for the Bi-dimensional Composition

3.1. Metamodel for the Vertical Composition

The methods defined in a *domain concept* are introduced for providing some *behavior* (see Fig. 5

Table 1. Different mappings of metamodel concepts on their instances at model level (for application development in Java and PHP respectively)

Domain	Product		Activity
Metamodel	ProductType		DataType
Model	(Java)	(PHP)	
	Use Case Document	Use Case Document	Requirement
	JML Specification	UML Specification	Specification
	Java File	PHP File	Program
	URL Bugzilla	Vision Project	BugReport

for the correspondent metamodel elements). In most cases, only a part (if any) of the *behavior* is implemented inside the method itself, because, most often, its functionality involves the execution of some tools. The notion of *Feature* has been defined to provide the code that contains one or more method *interceptions* and calls the services that actually implement the expected *behavior* of that methods. Additionally, a *feature* can implement a concern attached to that method, like security or persistency, which can be an optional *behavior*, as in product line approaches.

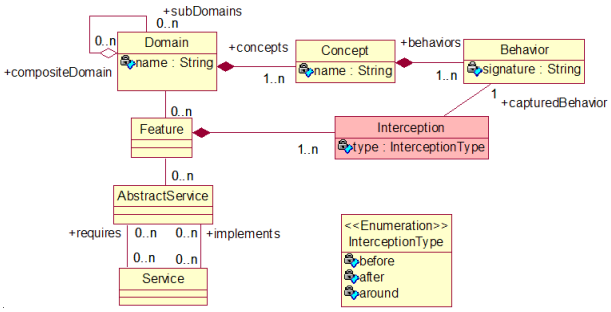


Figure 5. Metamodel for the vertical composition

For the vertical composition, the non-homogeneous units of reuse correspond to the generic notion of *Service* (see Fig. 5). At instantiation, they may correspond to components, tools, COTS etc. In our *Product* domain, the persistency *service* may be supplied either by SQL storage, or by a repository of another versioning system, like Subversion or CVS; the choice can be done by the client. For the example from Figure 4, the method *getProducts* of the class *ProductType* is empty and it is its associated *feature* that delegates the call to a database where actual products are stored. However, a *feature* is not related directly to *services*, but through *abstract services* – an abstraction for a set of functionalities defined in a Java interface, which are ultimately executed by components/tools representing the *services* (i.e. implementing its methods).

More than one *feature* can be attached to the same method and each *feature* can address a different concern. The word *feature* is used in the product line approach to express a possible variability that may be attached to a concept. Our approach is a combination of the product line intention with the AOP implementation.

Moreover, the purpose is to aid software engineers as much as possible, in the design and development of such kind of applications. By using the Codèle tool, which "knows" the metamodel from Figure 5, the software engineer simply creates instances of its concepts (*Behavior, Interception, Feature, Service* etc.) and the tool generates the corresponding code in the Eclipse framework. As well as all Mélusine DSLs, Codèle metamodels are implemented with Java, whereas AspectJ, its aspect-oriented extension, is used for delegating the implementation to different tools and/or components (instances of the *Service* concept).

3.2. Metamodel for the Horizontal Composition

In other similar approaches, as in model collaboration [26], AOP was mentioned as a possible solution for implementing collaboration templates in service oriented architectures (SOA), orchestration languages or coordination languages. As our approach is based on establishing relationships, it can also be compared to [1], where the properties of AOP concepts are identified (e.g. behavioral and structural cross-cutting advices, static and dynamic weaving). Our intention is to

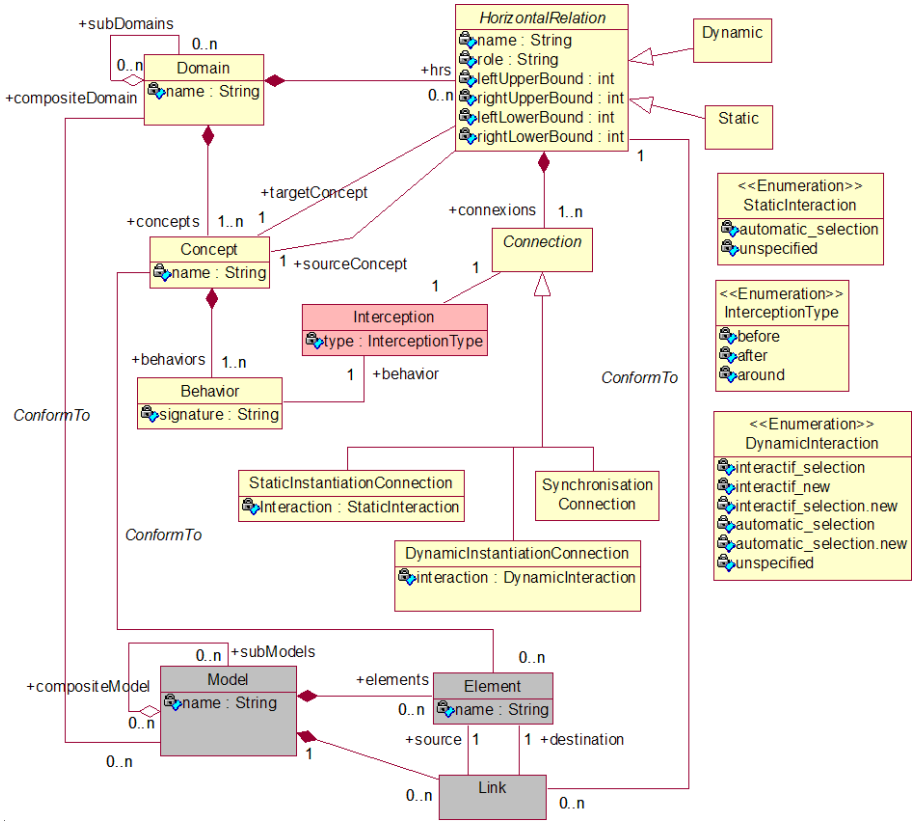


Figure 6. Metamodel for the horizontal composition

identify such properties at a more abstract level, such as aspects only constitute an implementation technique. The method we use for defining and generating horizontal compositions between domains is similar to transforming UML associations into Java [14], but using AOP, because we are not allowed to change the domain code.

To provide an effective support for domain composition, Mélusine requires a specific formal definition and semantics. The metamodel from Fig. 6 shows that domain composition relies on *Horizontal Relationship*, made of *connections*. A relationship not only represents a set of communication links between instances, but also expresses the interaction between them. The execution of an operation from an instance pertaining to one side of the relationship has consequences on the instances from the other side. In Codèle, such a piece of interaction is called *connection* and is established between a *source concept*, pertaining to the source *domain*, and a *destination concept* in the target *domain*. First, a connection performs the *interception* of the *behavior* (method) pertaining to the *source concept*, and then some computation, depending on its type: *Synchronization, StaticInstantiation, DynamicInstantiation*. Since *concepts* are reified as classes and operations are defined as their methods, a *connection* may be expressed based on the AOP mechanism: the method from one side is captured, allowing for the interaction with the methods from the other side. As a *concept* may have many methods, each one being able to participate to one or many *connection*(s), a *horizontal relationship* may manage many *connections*.

3.2.1. Composition Specific Semantics

From the experience gained while defining connections, some composition templates have been identified, such that some types of connections may be generalized and generated automatically. Connections are categorized according to their purpose:

– *Synchronization* – the most popular kind of connections, modifying the state of the in-

stance at the destination end, with respect to the changes performed for the instance at the source end;
- *Instantiation* – in charge of creating an instance of the horizontal relationship (a link between elements of the models to be composed) and, eventually, also with the creation of the instance at the destination end.

For establishing a link between two instances participating in a horizontal relationship, two issues must be considered: 1) the moment of creating the link, and 2) the alternatives for setting the destination instance. These semantics are taken into account when instantiating HRs at model level (see Fig. 4).

1) *The moment of creating the link.* Most often, a link is established when creating the instance that must be the origin of the link. These instances (representing elements of the models to be composed) are created either before execution (if they conform to AS concepts and are part of a domain specific model defined for a domain to be composed) or during the execution (if they conform to concepts introduced for interpreting these models). Therefore, it is possible to establish links either before or during the execution; the two situations actually correspond to the two types of horizontal relationships: *static* and *dynamic* respectively. For doing so, the method for creating the source instance (e.g. the constructor) is captured by the AOP machine and extended with the creation or the reification of the link; for the links defined before execution (between elements of domain specific models of the domains to be composed) the link is reified when the model elements are reified, just before starting the execution. In our example from Fig. 4, the links created between models (i.e. before the execution) are called *static*, while the links created to relate these models at execution are called *dynamic*. For example, the link between *Specification* and *JML Specification* is static, whereas the link between *DATA_0097* and *VERSION-0050* is dynamic.

2) *The alternatives for setting the destination instance.* For deciding the link destination end, there are two kinds of mapping functions:
- *Creation* (returning a *new* instance) and
- *Selection* (returning an existing instance).

Either to create or to select a destination instance, one should define some criteria, often based on the properties of the source instance. For example, the mapping function may create a destination instance, providing the source name as parameter (creation mapping) or it may look for the destination instance with the same name as the source instance (selection mapping). Besides the two alternatives above, the mapping function may adopt two kinds of processes:
- *Automatic:* the destination element is found automatically, if the searching criterion is provided;
- *Interactive:* the destination element is found with human intervention, if the searching criterion is not provided.

For the *Automatic* case, by default, Codèle supports a searching criterion based on a key attribute, like *name* or *identifier*. The default criterion is used if no user-defined searching criterion is provided.

The combination of the mapping kinds and processes presented above gives diverse ways to set the destination instance and the dynamic interaction may follow several valid possibilities, as also presented in the metamodel from Fig. 6:
- *Automatic.New*: the mapping function automatically creates and returns a new instance;
- *Automatic.Selection*: the mapping function automatically returns an existing instance;
- *Automatic.Selection.New*: the mapping function automatically searches for an existing instance and, if not found, creates a new one;
- *Interactive.Selection*: the destination instance is selected by a human, and
- *Interactive.Selection.New*: first, a human tries to select an existent destination instance; if he or she does not find anything appropriate, it is possible to ask for the creation of a new one.

The above options may be valid or not. If a link is created at execution time, all the above options may be used for setting the link destination. However, if a link is created before execution, the only valid option is *Auto-*

matic.Selection, because the link already exists and it must be simply reified.

4. Implementation Issues

4.1. Implementation Choices

Our approach follows the language/interpreter technology. However, to be later composable with other domain interpreters, the DSL interpreter must follow conventions in the way the concepts defined in the metamodel are mapped to the target programming language.

First, the target implementation language must be able to express the DSL operational semantics. Since the metamodels are object-oriented, it is convenient to use an object-oriented programming language, like Java or C++, or an executable metamodeling language, like Kermeta [25] or XMF (eXecutable Metamodelling Facility) [6]. Executable metamodeling languages allow not only the description of the model structure (the abstract syntax), but also of the behavior. Second, each concept in the metamodel must be mapped to one class in the target implementation language. Third, the target implementation language must provide support for aspect programming, to allow inserting the code responsible for the composition semantics into the original metamodel implementation (the set of corresponding classes, responsible for model interpretation) without changing the interpreter. In this context, one decided to use Java for implementing our interpreters, together with its aspect-oriented extension, AspectJ.

Models, defined using the DSL abstract syntax concepts, are technically reified as Java classes and then interpreted. This implies that the model is created before execution, while the instances of semantic domain concepts are only created during the execution. More precisely, at design time, the modeler only needs the abstract syntax concepts for creating a model – referred to as domain specific model; he or she does not need to be aware of the concepts related to the interpretation. Models are represented, at execution, as instances of the AS classes, and are interpreted using the semantic domain. At run time, the model is simply reified as instances of the interpreter classes and then interpreted. However, during execution, the interpreter modifies/creates/deletes instances of the abstract syntax concepts, and also creates instances of the DSL concepts corresponding to the semantic domain.

4.2. Code Generation

The Eclipse mappings currently used in Mélusine environment for the vertical composition are presented in Table 2. Actually, users never see, and even ignore, that AspectJ code is generated; for instance, they do not create an AspectJ project, but simply define and generate a feature associated with a concept. A similar idea is presented in [30], where Xtend and Xpand languages are used for specifying mappings from problem to solution spaces and the code generation is considered to be less error-prone than the manual coding.

To implement horizontal relationships in AspectJ, each horizontal relationship is also transformed into an AspectJ code. The mappings towards Eclipse artifacts used for Mélusine horizontal composition are indicated in Table 3.

4.3. Codèle Tool

This section introduces Codèle, as an implementation for the composition methodology previously presented. For supporting domain composition, we have developed the Codèle toolbox, in which dedicated editors allow one to: (i) Define horizontal relationships, (ii) Use horizontal relationships to define static model composition, (iii) Use horizontal relationships to define dynamic model composition.

From this information, Codèle automatically generates AspectJ captures and the code that implements the composition strategy. Implementing horizontal relationships in AspectJ is simple. Each connection is transformed into an AspectJ code that calls a method in a class gen-

Table 2. Mapping on Eclipse artifacts for the vertical composition metamodel

Metamodel element	**Eclipse artifact**	**Elements generated inside the artifacts**
Domain	Project	Interfaces for the domain management
Concept	Class	Skelton for the methods
Behavior	Method	Empty body by default
Feature	AspectJ Project	The AspectJ aspect and a class for the behavior
Abstract service	Project	Java interface defining the service interface
Service	Project	An interface and an implementation skeleton
Interception	AspectJ Capture	The corresponding AspectJ code

Table 3. Mapping between horizontal composition concepts and Eclipse artifacts

Metamodel element	**Eclipse artifact**	**Elements generated inside the artifacts**
Domain	Project	Predefined interfaces and classes
Concept	Class	None
Behavior	Method	None
HorizontalRelationship	AJ Class and Java classes	– an AspectJ file containing the code for all the interceptions – a Java file for each instantiation connections – a Java file for each synchronization connections
Interception	AspectJ Capture	Lines in the AspectJ file for the interception, and a Java file for the connection code

erated by Codèle; users never "see" it. In practice, the code for horizontal relationships semantics represents about 15% of the total code.

Under a unified graphical interface, Codèle implements different subsystems:

– *Relationships Editor*, which is responsible to create horizontal relationships, according to the properties presented above; see an example in Fig. 7, for defining a relationship between *DataType* from *Activity* domain, and *ProductType* from *Product* domain;
– *Captures Generator*, which generates AspectJ code, and creates a Java class in which the user can define the connection semantics;
– *Dynamic Model Composition Editor*, for dynamic link creation and life cycle;
– *Static Model Composition Editor* for the composition of two models, in their abstract form; see an example in Fig. 8.

In our example, the *DataType* – *ProductType* horizontal relationship has been selected, for which one displays the corresponding instances, like Specification in the Activity domain, and JMLSpecification or JavaFile in the Product domain. As this horizontal relationship has been declared Static, the developer is asked to provide the pairs of model entities that must be linked together, according to that horizontal

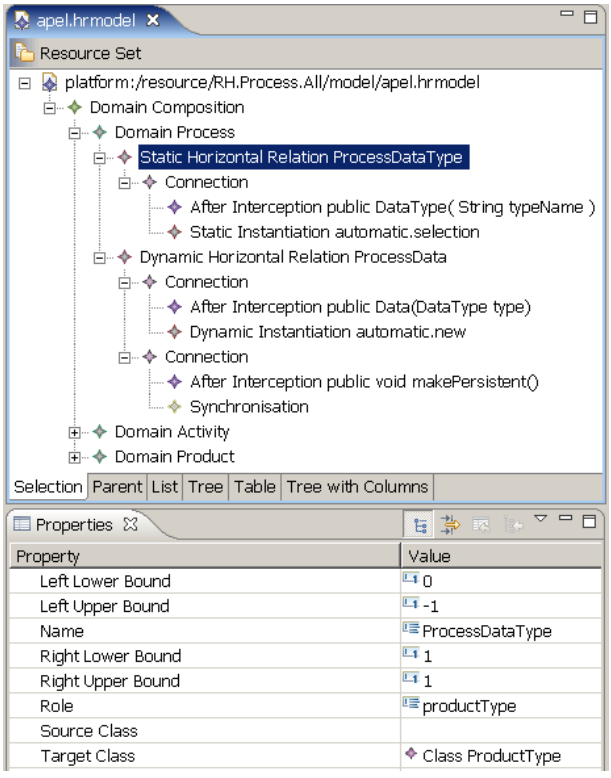


Figure 7. Defining horizontal relationships at metamodel level

relationship. Otherwise, they would have been selected automatically, at run time. The bottom panel lists the pairs that have been defined. For example, the data type called Program in the

Activity domain is now related to JavaFile in the Product domain. The system finds this information by introspecting the models and is in charge of creating these relationships at model level.

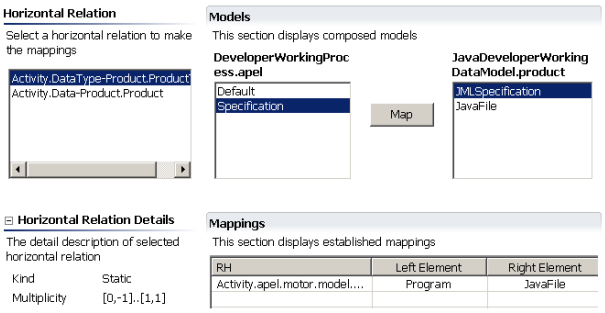


Figure 8. Defining static links at model level

Mélusine system, including the domain composition technology, was developed in 2000 and was used in a number of applications, both academic and industrial. A little less than one million lines were developed for this system, and dozens of domain compositions were performed. The work reported in this paper started with an analysis of these domain compositions, with the goal to find recurring concepts and patterns, and ended in the development of the Codèle tool. Since then, Codèle is integrated in different environments. In some environments, like FOCA [27] where domains are manually composed, Codèle is used only for model composition. In other systems, like Mélusine, Codèle is fully used, on a daily basis.

5. Evaluation of the Composition Approach

5.1. Related Works

The works on model/metamodel composition can be classified according to several criteria: the composition mechanism and the theme of research. According to the composition mechanism, these works could be split in two major categories: the heavyweight composition mechanism, which consists in model and metamodel merging [29], [23], [20]; and the lightweight mechanism, which involves establishing synchronization relationships [11] or weaving two models/metamodels, without changing their structures. The second mechanism is interesting because it is possible to compose models and metamodels and still use their existent tools.

According to the theme of research, model/metamodel composition is approached in three major areas: *Model Management, Aspect Oriented Modeling* and *Metamodeling.*

Model Management is a topic born in the MDE (Model Driven Engineering) context. This community is interested in platforms manipulating and managing models, focusing on generic operators to be applied on models, which can be divided in three groups:

- *match* [3, 4], *relate* [21], *compare* [20] – for discovering correspondences between models;
- *merge* [21], [20], [7], *compose* [3], *weaving* [7] – for integrating models and
- *sewing* [7] – for relating models without changing their structure.

Several platforms have been developed, like AMMA [28], Rondo [10], EOL [23] and MOMENT [19]. One can qualify these *Model Management* approaches as heavyweight.

Aspect Oriented Modeling (AOM) applies the separation of concern principle of AOP in the modeling phase. Weaving consists in composing aspect models to a base model. The relationship between aspect model and base model is relative. A model can be both an aspect and the base; thus, two kinds of weaving have been identified: aspect/base weaving (called asymmetric), and base/base weaving (called symmetric). The first one is borrowed from AOP and usually uses a lightweight composition mechanism, while the second one is inspired from SOP (Subject Oriented Programming) and uses a heavyweight mechanism. Theme/UML [7] is an approach merging both kinds of weaving; the composition between the base models (called subject) is done with two kinds of composition relationships: *merge* or *override. Merge* integrates a subject with another one, while *override* replaces an existing subject with a new one. In all cases, these strategies change the composed model structure. The aspects in Theme/UML are designed in terms of aspect templates.

Metamodelling also treats model compositions, supported by *metamodeling* environments, like XMF (eXecutable Metamodelling Facility) [6] or GME (Generic Modeling Environment) [9]. XMF has a purpose that is similar to ours – lightweight model composition consisting of composing and executing models conforming to different metamodels. This is possible through synchronized mappings, written in XSync – a specific language of XMF, based on actions. Unfortunately, the metamodels also have to be written in a specific language – XCore, which is an extension of MOF. Therefore, we would not be able to reuse our metamodels (implemented in Java) nor our models, nor use AOP technique – which is a central requirement for model and metamodel reuse.

GME environment also supports the composition of models conforming to the same metamodel (using so-called references) and to different metamodels (using union and inheritance). However, it allows the creation of a composite metamodel, which may be used for defining new models; there is no possibility to reuse the existing models "as-is" and to keep the metamodels unchanged – an important requirement for our domain composition approach.

The canonical scheme for model composition proposed in [5] uses a weaving model, consisting in correspondences between model elements. Then, several transformations based on ATL (ATLAS Transformation Language) are used for obtaining the composite model. The composition semantics resides in these transformations. The weaving model may also be extended for creating a specific composition, using AMW (Atlas Model Weaver). This facility could be used for defining our horizontal relationships; however, our purpose was to obtain a composition tool based on wizards, which is easier to learn and only contains the concepts specific for our composition approach.

5.2. Specificities for Mélusine Composition

In order to make the domain composition task as simple as possible, the metamodels presented above took into account the specificities of Mélusine domains. Consequently, the composition we realized is specific for this situation, as opposed to other approaches, which try to provide mechanisms for composing heterogeneous models in general contexts, generally without specifying how to implement them precisely.

The technique used at each composition level is different. At code level one uses AOP technique; at model and metamodel levels one establishes relationships. The main reason for this choice was to compose domains without changing their models or the associated tools and environments.

The elaboration of metamodels that support code generation in Codèle tool was possible after years of performing Mélusine domain compositions. This experience also led to the definition of a methodology for developing horizontal relationships, described in [11]. Moreover, through trials and errors, one found recurring patterns of code for defining vertical and horizontal relationships and it was possible to identify some of their functional and non functional characteristics. Codèle embodies and formalizes this knowledge through simple panels, such that users "only" need to write code for the non standard functionalities. Practice showed that, in average, more than half of the code is generated, in an error prone manner, managing the low level technical code – including AOP captures, aspect generation and so on. The user's added code fully ignores the generated one and the existence of AOP; it describes the added functionality at the logical level. Experience with Codèle has shown a dramatic simplification for writing relationships, and the elimination of the most difficult bugs; there are also some cases where the generated code was sufficient, allowing application composition without any programming.

However, many other non-functional characteristics could be identified and generated in the same way, and Codèle can (should) be extended to support them. We have also discovered that some, if not most, non-functional characteristics cannot be defined as a domain (security, performance, transaction etc.), and therefore these non-functional properties cannot be added

through horizontal relationships. For these properties, we have developed another technique, called model annotation, described in [27].

6. Conclusion

The division of applications in parts can be performed by reusing large functional areas, called domains, which are primary elements for dividing the problem in parts, and atoms on which our composition technique is applied.

A domain is usually implemented by reusing existing parts, found on the market or inside the company, which are components or tools of various size and nature. We call vertical composition the technique which consists in relating the abstract elements found in the domain model, with the existing components found in the company. Reuse imposes that vertical relationships are implemented, without changing the domain concepts, or the existing components. In our approach, one develops independent and autonomous domains, which become the primary units for reuse, whose interfaces are their domain models (DSLs).

Domain composition is performed by composing their DSLs, without any change in their abstract syntax or semantics. This is called horizontal composition, defining relationships between modeling elements pertaining to the composed domains. In this way, the tools/environments in charge of editing, analyzing and executing the models, as well as the knowledge of practitioners, are kept unchanged. Tools, environments and models can be reused "as-is" and thus they can continue to be used by the existing applications that rely on them, which is a critical property in real operational contexts.

An important goal of our approach was to raise the level of abstraction and the granularity level at which large applications are designed, decomposed and recomposed. Moreover, these large elements are highly reusable, because the composition only needs to "see" their abstract models, not their implementation. Finally, by relating domain concepts using wizards, most compositions can be performed by domain experts, not necessarily by highly trained technical experts, as it would be the case if directly using AOP techniques.

References

[1] E. Barra Zavaleta, G. Génova Fuster, and J. Llorens Morillo. An approach to aspect modelling with uml 2.0. In *Proceedings of the UML 2004 Workshop on Aspect-Oriented modeling*, Lisbon, Portugal, 2004.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] P. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, Asilomar, CA, USA, 2003.

[4] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, Shanghai, China, 2006. ACM.

[5] J. Bézivin, S. Bouzitouna, M. D. D. Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige. A canonical scheme for model composition. In *Proceedings of the European Conference in Model Driven Architecture (EC-MDA)*, Bilbao, Spain, 2006.

[6] T. Clark and al. Applied metamodelling – a foundation for language driven development version 0.1. Xactium, Editor, 2004.

[7] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002.

[8] T. Dave. Reflective software engineering – from MOPS to AOSD. *Journal of Object Technology*, 1(4), 2002.

[9] J. Davis. GME: the generic modeling environment. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '03)*, Anaheim, CA, USA, 2003.

[10] M. Didonet Del Fabro and F. Jouault. Model transformation and weaving in the amma platform. In *Proceedings of the Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE)*, Braga, Portugal, 2005.

[11] J. Estublier, A. D. Ionita, and G. Vega. Relationships for domain reuse and composition.

Journal of Research and Practice in Information Technology, 38(4):135–162, 2006.

[12] J. Estublier, G. Vega, and A. Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Proceedings of the MoDELS/UML Conference*, pages 69–83, Jamaica, 2005. Lecture Notes in Computer Science.

[13] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, ISBN10: 0321219767, 2004.

[14] G. Génova, C. Ruiz del Castillo, and J. Lloréns. Mapping UML associations into Java code. *Journal of Object Technology*, 2(5):135–162, 2003.

[15] J. Gray, T. Bapty, S. Neema, D. Schmidt, A. Gokhale, and N. B. An approach for supporting aspect-oriented domain modeling. In *Proceedings of GPCE*. LNCS 2830, Springer Verlag, 2003.

[16] W. Ho, J.-M. Jezequel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect-oriented UML designs. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, pages 99–105, Enschede, The Netherlands, 2002.

[17] A. D. Ionita, J. Estublier, and G. Vega. Variations in model-based composition of domains. In *Proceedings of the Software and Service Variability Management Workshop*, Helsinki, Finland, April 2007.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[19] D. Kolovos, R. Paige, and F. Polack. Eclipse development tools for epsilon. In *Proceedings of the Eclipse Summit Europe, Eclipse Modeling Symposium*, Esslingen, Germany, 2006.

[20] D. Kolovos, R. Paige, and F. Polack. Merging models with the epsilon merging language (eml). In *Proceedings of MoDELS'06*, pages 215–229. LNCS 4199, 2006.

[21] I. Kurtev and M. Didonet Del Fabro. A DSL for definition of model composition operators. In *Proceedings of the Models and Aspects Workshop at ECOOP*, Nantes, France, 2006.

[22] T. Le-Anh, J. Estublier, and J. Villalobos. Multi-level composition for software federations. In *Proceedings of the SC'2003 Conference*, Warsaw, Poland, April (2003). IEEE Computer Society Press.

[23] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *Proceedings of the International Conference on Special Interest Group on Management of Data (SIGMOD)*, San Diego, California, June, 2003.

[24] M. Monga. Aspect-oriented programming as model driven evolution. In *Proceedings of the linking aspect technology and evolution (LATE) workshop*, Chicago, 2005.

[25] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the MoDELS/UML Conference*, Jamaica, 2005. Lecture Notes in Computer Science.

[26] A. Occello, O. Casile, A. Dery-Pinna, and M. Riveill. Making domain-specific models collaborate. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, Montréal, Canada, 2007.

[27] G. Pedraza and J. Estublier. An extensible service orchestration framework through concern composition. intl workshop on non-functionnal properties in domain specific languages. In *Proceedings of the NFPDML conference*, Toulouse France, 2008.

[28] T. Reiter and al. Model integration through mega operations. In *Proceedings of the Workshop on Model-driven Web Engineering (MDWE)*, Sydney, 2005.

[29] M. Sabetzadeh and S. Easterbrook. Easterbrook: An algebraic framework for merging incomplete and inconsistent views. In *Proceedings of the 13th IEEE International Requirements Engineering Conference*, pages 306–318, 2005.

[30] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Prouct Line Conference (SPLC)*, Kyoto, Japan, 2007.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Aspect-Oriented Change Realizations and Their Interaction

Valentino Vranić*, Radoslav Menkyna*, Michal Bebjak*, Peter Dolog**
*Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Slovakia
**Department of Computer Science, Aalborg University, Denmark
vranic@fiit.stuba.sk, radu@ynet.sk, mbebjak@gmail.com, dolog@cs.aau.dk

Abstract

With aspect-oriented programming, changes can be treated explicitly and directly at the programming language level. An approach to aspect-oriented change realization based on a two-level change type model is presented in this paper. In this approach, aspect-oriented change realizations are mainly based on aspect-oriented design patterns or themselves constitute pattern-like forms in connection to which domain independent change types can be identified. However, it is more convenient to plan changes in a domain specific manner. Domain specific change types can be seen as subtypes of generally applicable change types. These relationships can be maintained in a form of a catalog. Some changes can actually affect existing aspect-oriented change realizations, which can be solved by adapting the existing change implementation or by implementing an aspect-oriented change realization of the existing change without having to modify its source code. As demonstrated partially by the approach evaluation, the problem of change interaction may be avoided to a large extent by using appropriate aspect-oriented development tools, but for a large number of changes, dependencies between them have to be tracked. Constructing partial feature models in which changes are represented by variable features is sufficient to discover indirect change dependencies that may lead to change interaction.

1. Introduction

Change realization consumes enormous effort and time during software evolution. Once implemented, changes get lost in the code. While individual code modifications are usually tracked by a version control tool, the logic of a change as a whole vanishes without a proper support in the programming language itself.

By its capability to separate crosscutting concerns, aspect-oriented programming enables to deal with change explicitly and directly at programming language level. Changes implemented this way are pluggable and — to the great extent — reapplicable to similar applications, such as applications from the same product line.

Customization of web applications represents a prominent example of that kind. In customization, a general application is being adapted to the client's needs by a series of changes. With each new version of the base application, all the changes have to be applied to it. In many occasions, the difference between the new and old application does not affect the structure of changes, so if changes have been implemented using aspect-oriented programming, they can be simply included into the new application build without any additional effort.

Even conventionally realized changes may interact, i.e. they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations. This is even more remark-

able in aspect-oriented change realization due to pervasiveness of aspect-oriented programming as such.

We have already reported briefly our initial work in change realization using aspect-oriented programming [1]. In this paper[1], we present our improved view of the approach to change realization based on a two-level change type model. Section 2 presents our approach to aspect-oriented change realization. Section 3 describes briefly the change types we have discovered so far in the web application domain. Section 4 discusses how to deal with a change of a change. Section 5 proposes a feature modeling based approach of dealing with change interaction. Section 6 describes the approach evaluation and outlooks for tool support. Section 7 discusses related work. Section 8 presents conclusions and directions of further work.

2. Changes as Crosscutting Requirements

A change is initiated by a change request made by a user or some other stakeholder. Change requests are specified in domain notions similarly as initial requirements are. A change request tends to be focused, but it often consists of several different — though usually interrelated — requirements that specify actual changes to be realized. By decomposing a change request into individual changes and by abstracting the essence out of each such change while generalizing it at the same time, a change type applicable to a range of the applications that belong to the same domain can be defined.

We will present our approach by a series of examples on a common scenario[2]. Suppose a merchant who runs his online music shop purchases a general affiliate marketing software [11] to advertise at third party web sites denoted as affiliates. In a simplified schema of affiliate marketing, a customer visits an affiliate's site which refers him to the merchant's site. When he buys something from the merchant, the provision is given to the affiliate who referred the sale. A general affiliate marketing software enables to manage affiliates, track sales referred by these affiliates, and compute provisions for referred sales. It is also able to send notifications about new sales, signed up affiliates, etc.

The general affiliate marketing software has to be adapted (customized), which involves a series of changes. We will assume the affiliate marketing software is written in Java, so we can use AspectJ, the most popular aspect-oriented language, which is based on Java, to implement some of these changes.

In the AspectJ style of aspect-oriented programming, the crosscutting concerns are captured in units called aspects. Aspects may contain fields and methods much the same way the usual Java classes do, but what makes possible for them to affect other code are genuine aspect-oriented constructs, namely: *pointcuts*, which specify the places in the code to be affected, *advices*, which implement the additional behavior before, after, or instead of the captured *join point* (a well-defined place in the program execution) — most often method calls or executions — and *inter-type declarations*, which enable introduction of new members into types, as well as introduction of compilation warnings and errors.

2.1. Domain Specific Changes

One of the changes of the affiliate marketing software would be adding a backup SMTP server to ensure delivery of the notifications to users. Each time the affiliate marketing software needs to send a notification, it creates an instance of the SMTPServer class which handles the connection to the SMTP server.

An SMTP server is a kind of a resource that needs to be backed up, so in general, the type of the change we are talking about could be denoted as *Introducing Resource Backup*. This change type is still expressed in a domain specific way. We can clearly identify a crosscutting concern of maintaining a backup resource that

[1] This paper represents an extended version of our paper presented at CEE-SET 2008 [28].
[2] This is an adapted scenario published in our earlier work [1].

has to be activated if the original one fails and implement this change in a single aspect without modifying the original code:

```
public class SMTPServerM extends SMTPServer {
 ...
}
 ...
public aspect SMTPServerBackupA {
  public pointcut SMTPServerConstructor(URL url,
                                        String user,
                                        String password):
    call(SMTPServer.new (..)) && args(url, user,
                                        password);
  SMTPServer around(URL url, String user,
                                        String password):
    SMTPServerConstructor(url, user, password)
  {
    return getSMTPServerBackup(proceed(url, user,
                                        password));
  }
  private SMTPServer
  getSMTPServerBackup(SMTPServer obj)
  {
    if (obj.isConnected()) {
      return obj;
    } else {
      return new SMTPServerM(obj.getUrl(),
                                        obj.getUser(),
                                        obj.getPassword());
    }
  }
}
```

The **around**() advice captures constructor calls of the SMTPServer class and their arguments. This kind of advice takes complete control over the captured join point and its return clause, which is used in this example to control the type of the SMTP server being returned. The policy is implemented in the getSMTPServerBackup() method: if the original SMTP server can't be connected to, a backup SMTP server class SMTPServerM instance is created and returned.

We can also have another aspect — say SMTPServerBackupB — intended for another application configuration that would implement a different backup policy or simply instantiate a different backup SMTP server.

2.2. Generally Applicable Changes

Looking at this code and leaving aside SMTP servers and resources altogether, we notice that it actually performs a class exchange. This idea can be generalized and domain details abstracted out of it bringing us to the *Class Exchange* change type [1] which is based on the Cuckoo's Egg aspect-oriented design pattern [20]:

```
public class AnotherClass extends MyClass {
 ...
}
 ...
public aspect MyClassSwapper {
  public pointcut myConstructors():
    call(MyClass.new ());
  Object around(): myConstructors()
  {
    return new AnotherClass();
  }
}
```

2.3. Applying a Change Type

It would be beneficial if the developer could get a hint on using the Cuckoo's Egg pattern based on the information that a resource backup had to be introduced. This could be achieved by maintaining a catalog of changes in which each domain specific change type would be defined as a specialization of one or more generally applicable changes.

When determining a change type to be applied, a developer chooses a particular change request, identifies individual changes in it, and determines their type. Figure 1 shows an example situation. Domain specific changes of the D1 and D2 type have been identified in the Change Request 1. From the previously identified and cataloged relationships between change types we would know their generally applicable change types are G1 and G2.

A generally applicable change type can be a kind of an aspect-oriented design pattern (consider G2 and AO Pattern 2). A domain specific change realization can also be complemented by an aspect-oriented design pattern (or several ones), which is expressed by an association between them (consider D1 and AO Pattern 1).

Each generally applicable change has a known domain independent code scheme (G2's code scheme is omitted from the figure). This code scheme has to be adapted to the context

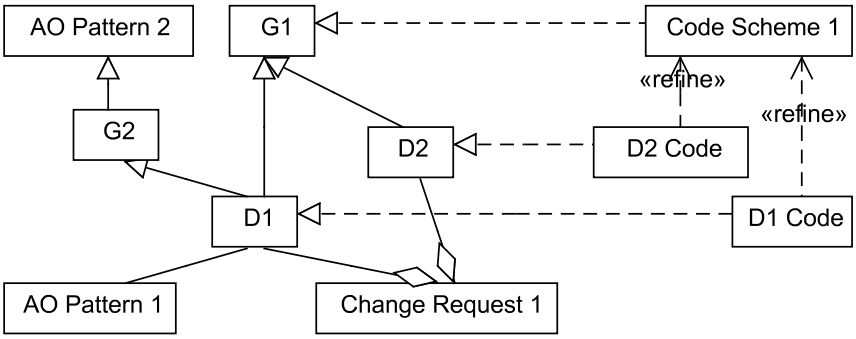


Figure 1. Generally applicable and domain specific changes

of a particular domain specific change, which may be seen as a kind of refinement (consider D1 Code and D2 Code).

3. Catalog of Changes

To support the process of change selection, the catalog of changes is needed in which the generalization–specialization relationships between change types would be explicitly established. The following list sums up these relationships between change types we have identified in the web application domain (the domain specific change type is introduced first):

- One Way Integration: Performing Action After Event,
- Two Way Integration: Performing Action After Event,
- Adding Column to Grid: Performing Action After Event,
- Removing Column from Grid: Method Substitution,
- Altering Column Presentation in Grid: Method Substitution,
- Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification,
- Removing Fields from Form: Additional Return Value Checking/Modification,
- Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event,
- Introducing User Rights Management: Border Control with Method Substitution,
- User Interface Restriction: Additional Return Value Checking/Modifications,
- Introducing Resource Backup: Class Exchange.

We have already described Introducing Resource Backup and the corresponding generally applicable change, Class Exchange. Here, we will briefly describe the rest of the domain specific change types we identified in the web application domain along with the corresponding generally applicable changes. The generally applicable change types are described where they are first mentioned to make sequential reading of this section easier. In a real catalog of changes, each change type would be described separately.

3.1. Integration Changes

Web applications often have to be integrated with other systems. Suppose that in our example the merchant wants to integrate the affiliate marketing software with the third party newsletter which he uses. Every affiliate should be a member of the newsletter. When an affiliate signs up to the affiliate marketing software, he should be signed up to the newsletter, too. Upon deleting his account, the affiliate should be removed from the newsletter, too.

This is a typical example of the *One Way Integration* change type [1]. Its essence is the one way notification: the integrating application notifies the integrated application of relevant events. In our case, such events are the affiliate sign-up and affiliate account deletion.

Such integration corresponds to the *Performing Action After Event* change type [1]. Since events are actually represented by methods, the desired action can be implemented in an after advice:

```
public aspect PerformActionAfterEvent {
  pointcut methodCalls(TargetClass t, int a):...;
  after( /* captured arguments */):
           methodCalls( /* captured arguments */)
  {
           performAction( /* captured arguments */);
  }
  private void performAction( /* arguments */)
  {
    /* action logic */
  }
}
```

The after advice executes after the captured method calls. The actual action is implemented as the performAction() method called by the advice.

To implement the one way integration, in the after advice we will make a post to the newsletter sign-up/sign-out script and pass it the e-mail address and name of the newly signed-up or deleted affiliate. We can seamlessly combine multiple one way integrations to integrate with several systems.

The *Two Way Integration* change type can be seen as a double One Way Integration. A typical example of such a change is data synchronization (e.g., synchronization of user accounts) across multiple systems. When a user changes his profile in one of the systems, these changes should be visible in all of them. In our example, introducing a forum for affiliates with synchronized user accounts for affiliate convenience would represent a Two Way Integration.

3.2. Introducing User Rights Management

In our affiliate marketing application, the marketing is managed by several co-workers with different roles. Therefore, its database has to be updated from an administrator account with limited permissions. A restricted administrator should not be able to decline or delete affiliates, nor modify the advertising campaigns and banners that have been integrated with the web sites of affiliates. This is an instance of the *Introducing User Rights Management* change type.

Suppose all the methods for managing campaigns and banners are located in the campaigns and banners packages. The calls to these methods can be viewed as a region prohibited to the restricted administrator. The Border Control design pattern [20] enables to partition an application into a series of regions implemented as pointcuts that can later be operated on by advices [1]:

```
pointcut prohibitedRegion():
  (within(application.Proxy)
  && call(void *.*(..)))
  || (within(application.campaigns.+)
  && call(void *.*(..)))
  || within(application.banners.+)
  || call(void Affiliate.decline(..))
  || call(void Affiliate.delete(..));
```

What we actually need is to substitute the calls to the methods in the region with our own code that will let the original methods execute only if the current user has sufficient rights. This can be achieved by applying the *Method Substitution* change type which is based on an around advice that enables to change or completely disable the execution of methods. The following pointcut captures all method calls of the method called method() belonging to the TargetClass class:

```
pointcut allmethodCalls(TargetClass t, int a):
  call(ReturnType TargetClass.method(..)) &&
  target(t) && args(a);
```

Note that we capture method calls, not executions, which gives us the flexibility in constraining the method substitution logic by the context of the method call. The **call**() pointcut captures all the calls of TargetClass.method(), the **target**() pointcut is used to capture the reference to the target object, and the method arguments (if we need them) are captured by an **args**() pointcut. In the example code, we assume method() has one integer argument and capture it with this pointcut.

The following example captures the method() calls made within the control flow of any of the CallingClass methods:

```
pointcut specificmethodCalls(TargetClass t, int a):
   call(ReturnType TargetClass.method(a))
   && target(t) && args(a)
   && cflow(call(* CallingClass.*(..)));
```

This embraces the calls made directly in these methods, but also any of the method() calls made further in the methods called directly or indirectly by the CallingClass methods.

By making an around advice on the specified method call capturing pointcut, we can create a new logic of the method to be substituted:

```
public aspect MethodSubstition {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(TargetClass t, int a):
                                    methodCalls(t, a) {
      if (. . .) {
         . . . } // the new method logic
      else
         proceed(t, a);
   }
}
```

3.3. User Interface Restriction

It is quite annoying when a user sees, but can't access some options due to user rights restrictions. This requires a *User Interface Restriction* change type to be applied. We have created a similar situation in our example by a previous change implementation that introduced the restricted administrator (see Sect. 3.2). Since the restricted administrator can't access advertising campaigns and banners, he shouldn't see them in menu either.

Menu items are retrieved by a method and all we have to do to remove the banners and campaigns items is to modify the return value of this method. This may be achieved by applying a *Additional Return Value Checking/Modification* change which checks or modifies a method return value using an around advice:

```
public aspect AdditionalReturnValueProcessing {
   pointcut methodCalls(TargetClass t, int a): . . .;
   private ReturnType retValue;
   ReturnType around():
               methodCalls(/* captured arguments */) {
      retValue = proceed(/* captured arguments */);
      processOutput(/* captured arguments */);
      return retValue;
   }
   private void processOutput(/* arguments */) {
      // processing  logic
   }
}
```

In the around advice, we assign the original return value to the private attribute of the aspect. Afterwards, this value is processed by the processOutput() method and the result is returned by the around advice.

3.4. Grid Display Changes

It is often necessary to modify the way data are displayed or inserted. In web applications, data are often displayed in grids, and data input is usually realized via forms. Grids usually display the content of a database table or collation of data from multiple tables directly. Typical changes required on grid are adding columns, removing them, and modifying their presentation. A grid that is going to be modified must be implemented either as some kind of a reusable component or generated by row and cell processing methods. If the grid is hard coded for a specific view, it is difficult or even impossible to modify it using aspect-oriented techniques.

If the grid is implemented as a data driven component, we just have to modify the data passed to the grid. This corresponds to the Additional Return Value Checking/Modification change (see Sect. 3.3). If the grid is not a data driven component, it has to be provided at least with the methods for processing rows and cells.

Adding Column to Grid can be performed *after an event* of displaying the existing columns of the grid which brings us to the Performing Action After Event change type (see Sect. 3.1). Note that the database has to reflect the change, too. *Removing Column from Grid* requires a conditional execution of the method that displays cells, which may be realized as a Method Substitution change (see Sect. 3.2).

Alterations of a grid are often necessary due to software localization. For example, in Japan and Hungary, in contrast to most other countries, the surname is placed before the given names. The *Altering Column Presentation in Grid* change type requires preprocessing of all the data to be displayed in a grid before actually displaying them. This may be easily achieved by modifying the way the grid cells are rendered,

which may be implemented again as a Method Substitution (see Sect. 3.2):

```
public aspect ChangeUserNameDisplay {
   pointcut displayCellCalls(String name, String value):
      call(void UserTable.displayCell(..))  ||
                        args(name, value);
   around(String name, String value):
                        displayCellCalls(name, value) {
      if (name ==
         "<the name of the column to be modified>") {
         . . . // display the modified column
      } else {
         proceed(name, value);
      }
   }
}
```

3.5. Input Form Changes

Similarly to tables, forms are often subject to modifications. Users often want to add or remove fields from forms or pose additional constraints on their input fields. Note that to be possible to modify forms using aspect-oriented programming they may not be hard coded in HTML, but generated by a method. Typically they are generated from a list of fields implemented by an enumeration.

Going back to our example, assume that the merchant wants to know the genre of the music which is promoted by his affiliates. We need to add the genre field to the generic affiliate sign-up form and his profile form to acquire the information about the genre to be promoted at different affiliate web sites. This is a change of the *Adding Fields to Form* type. To display the required information, we need to modify the affiliate table of the merchant panel to display genre in a new column. This can be realized by applying the Enumeration Modification change type to add the genre field along with already mentioned Additional Return Value Checking/Modification in order to modify the list of fields being returned (see Sect. 3.3).

The realization of the *Enumeration Modification* change type depends on the enumeration type implementation. Enumeration types are often represented as classes with a static field for each enumeration value. A single enumeration value type is represented as a class with a field that holds the actual (usually integer) value and its name. We add a new enumeration value by introducing the corresponding static field:

```
public aspect NewEnumType {
   public static EnumValueType
                        EnumType.NEWVALUE =
      new EnumValueType(10, "<new value name>");
}
```

The fields in a form are generated according to the enumeration values. The list of enumeration values is typically accessible via a method provided by it. This method has to be addressed by an Additional Return Value Checking/Modification change.

For *Removing Fields from Form*, an Additional Return Value Checking/Modification change is sufficient. Actually, the enumeration value would still be included in the enumeration, but this would not affect the form generation.

If we want to introduce additional validations on form input fields in an application without a built-in validation, which constitutes an *Introducing Additional Constraint on Fields* change, an *Additional Parameter Checking* change can be applied to methods that process values submitted by the form. This change enables to introduce an additional validation or constraint on method arguments. For this, we have to specify a pointcut that will capture all the calls of the affected methods along with their context similarly as in Sect. 3.2. Their arguments will be checked by the check() method called from within an around advice which will throw WrongParamsException if they are not correct:

```
public aspect AdditionalParameterChecking {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(/* arguments */) throws
                        WrongParamsException:
      methodCalls(/* arguments */) {
      check(/* arguments */);
      return proceed(/* arguments */);
   }
   void check(/* arguments */) throws
                        WrongParamsException {
      if (arg1 != <desired value>)
         throw new WrongParamsException();
   }
}
```

Adding a new validator to an application that already has a built-in validation is realized

by simply including it in the list of validators. This can be done by implementing the Performing Action After Event change (see Sect. 3.1), which would add the validator to the list of validators after the list initialization.

4. Changing a Change

Sooner or later there will be a need for a change whose realization will affect some of the already applied changes. There are two possibilities to deal with this situation: a new change can be implemented separately using aspect-oriented programming or the affected change source code could be modified directly. Either way, the changes remain separate from the rest of the application.

The possibility to implement a change of a change using aspect-oriented programming and without modifying the original change is given by the aspect-oriented programming language capabilities. Consider, for example, advices in AspectJ. They are unnamed, so can't be referred to directly. The primitive pointcut **adviceexecution**(), which captures execution of all advices, can be restricted by the **within**() pointcut to a given aspect, but if an aspect contains several advices, advices have to be annotated and accessed by the **@annotation**() pointcut, which was impossible in AspectJ versions that existed before Java was extended with annotations.

An interesting consequence of aspect-oriented change realization is the separation of crosscutting concerns in the application which improves its modularity (and thus makes easier further changes) and may be seen as a kind of aspect-oriented refactoring. For example, in our affiliate marketing application, the integration with a newsletter — identified as a kind of One Way Integration — actually was a separation of integration connection, which may be seen as a concern of its own. Even if these once separated concerns are further maintained by direct source code modification, the important thing is that they remain separate from the rest of the application. Implementing a change of a change using aspect-oriented programming and without modifying the original change is interesting mainly if it leads to separation of another crosscutting concern.

5. Capturing Change Interaction by Feature Models

Some change realizations can *interact*: they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations. With increasing number of changes, change interaction can easily escalate into a serious problem: serious as *feature* interaction.

Change realizations in the sense of the approach presented so far actually resemble features as coherent pieces of functionality. Moreover, they are virtually pluggable and as such represent *variable* features. This brings us to feature modeling as an appropriate technique for managing variability in software development including variability among changes. This section will show how to model aspect-oriented changes using feature modeling.

5.1. Representing Change Realizations

There are several feature modeling notations [26] of which we will stick to a widely accepted and simple Czarnecki–Eisenecker basic notation [5]. Further in this section, we will show how feature modeling can be used to manage change interaction with elements of the notation explained as needed.

Aspect-oriented change realizations can be perceived as variable features that extend an existing system. Fig. 2 shows the change realizations from our affiliate marketing scenario a feature diagram. A feature diagram is commonly represented as a tree whose root represents a concept being modeled. Our concept is our affiliate marketing software. All the changes are modeled as optional features (marked by an empty circle ended edges) that can but do not have to be included in a feature configuration — known also as concept instance — for it to be

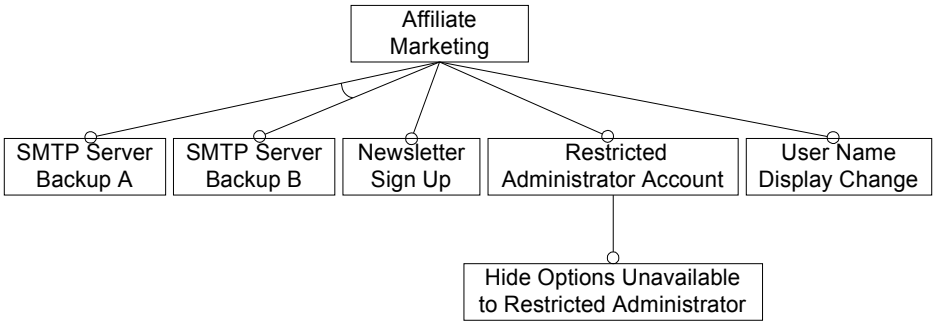


Figure 2. Affiliate marketing software change realizations in a feature diagram

valid. Recall adding a backup SMTP server discussed in Sect. 2.1. We considered a possibility of having another realization of this change, but we don't want both realizations simultaneously. In the feature diagram, this is expressed by alternative features (marked by an arc), so no Affiliate Marketing instance will contain both SMTP Server Backup A and SMTP Server Backup B.

A change realization can be meaningful only in the context of another change realization. In other words, such a change realization requires the other change realization. In our scenario, hiding options unavailable to a restricted administrator makes sense only if we introduced a restricted administrator account (see Sect. 3.3 and 3.2). Thus, the Hide Options Unavailable to Restricted Administrator feature is a subfeature of the Restricted Administrator Account feature. For a subfeature to be included in a concept instance its parent feature must be included, too.

5.2. Identifying Direct Change Interactions

Direct change interactions can be identified in a feature diagram with change realizations modeled as features of the affected software concept. Each dependency among features represents a potential change interaction. A direct change interaction may occur among alternative features or a feature and its subfeatures: such changes may affect the common join points. In our affiliate marketing scenario, alternative SMTP backup server change realizations are an example of such changes. Determining whether changes really interact requires analysis of dependant feature semantics with respect to the implementation of the software being changed. This is beyond feature modeling capabilities.

Indirect feature dependencies may also represent potential change interactions. Additional dependencies among changes can be discovered by exploring the software to which the changes are introduced. For this, it is necessary to have a feature model of the software itself, which is seldom the case. Constructing a complete feature model can be too costly with respect to expected benefits for change interaction identification. However, only a part of the feature model that actually contains edges that connect the features under consideration is needed in order to reveal indirect dependencies among them.

5.3. Partial Feature Model Construction

The process of constructing partial feature model is based on the feature model in which aspect-oriented change realizations are represented by variable features that extend an existing system represented as a concept (see Sect. 5.1).

The concept node in this case is an abstract representation of the underlying software system. Potential dependencies of the change realizations are hidden inside of it. In order to reveal them, we must factor out concrete features from the concept. Starting at the features that represent change realizations (leaves) we proceed bottom up trying to identify their parent features until related changes are not grouped in common subtrees. Figure 3 depicts this process.

The process will be demonstrated on YonBan, a student project management system de-

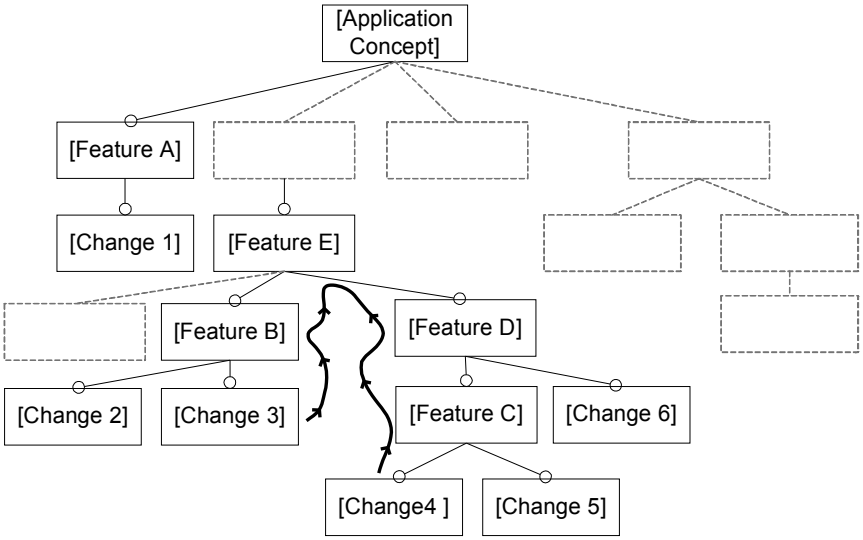


Figure 3. Constructing a partial feature model

veloped at Slovak University of Technology. We will consider the following changes in YonBan and their respective realizations indicated by generally applicable change types:

- Telephone Number Validating (realized as Performing Action After Event): to validate a telephone number the user has entered;
- Telephone Number Formatting (realized as Additional Return Value Checking/Modification): to format a telephone number by adding country prefix;
- Project Registration Statistics (realized as One Way Integration): to gain statistic information about the project registrations;
- Project Registration Constraint (realized as Additional Parameter Checking/Modification): to check whether the student who wants to register a project has a valid e-mail address in his profile;
- Exception Logging (realized as Performing Action After Event): to log the exceptions thrown during the program execution;
- Name Formatting (realized as Method Substitution): to change the way how student names are formatted.

These change realizations are captured in the initial feature diagram presented Fig. 4. Since there was no relevant information about direct dependencies among changes during their specification, there are no direct dependencies among the features that represent them either. The concept of the system as such is marked as open (indicated by square brackets), which means that new variable subfeatures are expected at it. This is so because we show only a part of the analyzed system knowing there are other features there.

Following this initial stage, we attempt to identify parent features of the change realization features as the features of the underlying system that are affected by them. Figure 5 shows such changes identified in our case. We found that Name Formatting affects the Name Entering feature. Project Registration Statistic and Project Registration Constraint change User Registration. Telephone Number Formatting and Telephone Number Validating are changes of Telephone Number Entering. Exception Logging affects all the features in the application, so it remains a direct feature of the concept. All these newly identified features are open because we are aware of the incompleteness of their subfeature sets.

We continue this process until we are able to identify parent features or until all the changes are found in a common subtree of the feature diagram, whichever comes first. In our example, we reached this stage within the following — and thus last — iteration which is presented in Fig. 6: we realized that Telephone Number Entering is a part of User Registration.

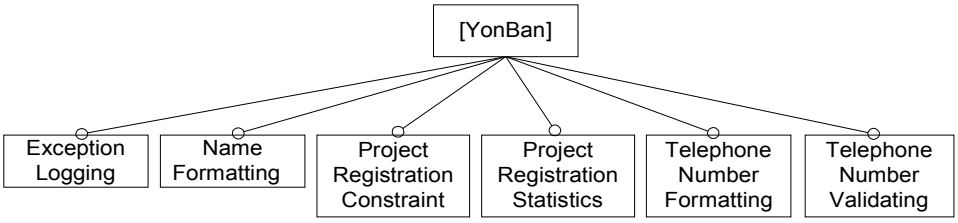


Figure 4. Initial stage of the YonBan partial feature model construction

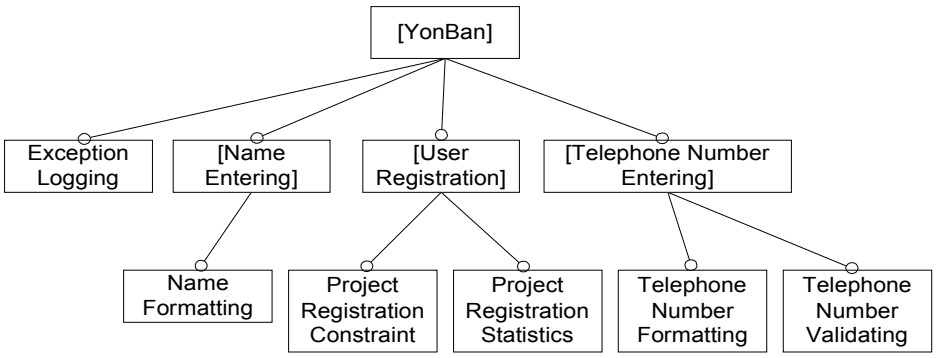


Figure 5. Identifying parent features in YonBan partial feature model construction

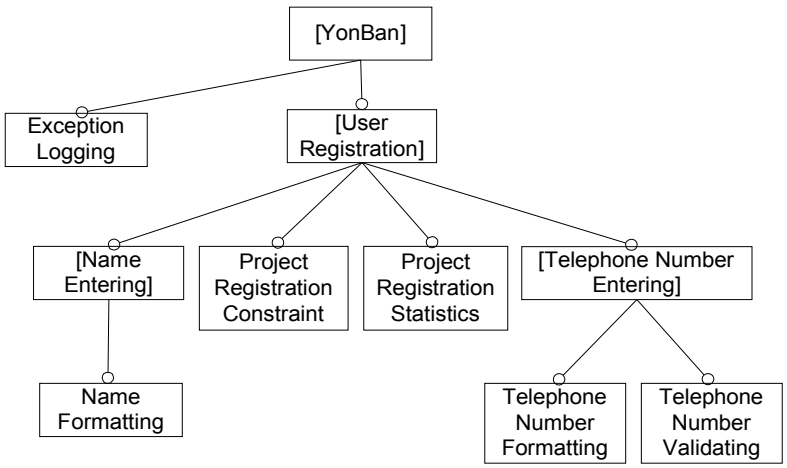


Figure 6. The final YonBan partial feature model

5.4. Dependency Evaluation

Dependencies among change realization features in a partial feature model constitute potential change realization interactions. A careful analysis of the feature model can reveal dependencies we have overlooked during its construction.

Sibling features (direct subfeatures of the same parent feature) are potentially interdependent. This problem can occur also among the features that are — to say so — indirect siblings, so we have to analyze these, too. Speaking in terms of change implementation, the code that implements the parent feature altered by one of the sibling change features can be dependent on the code altered by another sibling change feature or vice versa. The feature model points us to the locations of potential interaction.

In our example, we have a partial feature model (recall Fig. 6) and we understand the way the changes should be implemented based on their type (see Sect. 5.3). Project Registra-

tion Constraint and Project Registration Statistic change are both direct subfeatures of User Registration. The two aspects that would implement these changes would advise the same project registration method, and this indeed can lead to interaction. In such cases, precedence of aspects should be set (in AspectJ, **dominates** inter-type declaration enables this). Another possible problem in this particular situation is that the Project Registration Constraint change can disable the execution of the project registration method. If the Project Registration Statistic change would use an **execution**() pointcut, everything would be all right. On the other hand, if the Project Registration Statistic change would use a **call**() pointcut, the registration statistic advice would be still executed even when the registration method would not be executed. This would cause an undesirable system behavior where also registrations canceled by Project Registration Constraint would be counted in statistic. The probability of a mistake when a **call**() pointcut is used instead of the **execution**() pointcut is higher if the Project Registration Statistic change would be added first.

Telephone Number Formatting and Telephone Number Validating are another example of direct subfeatures. In this case, the aspects that would implement these changes apply to different join points, so apparently, no interaction should occur. However, a detailed look uncovers that Telephone Number Formatting change alters the value which the Telephone Number Validating change has to validate. This introduces a kind of logical dependency and to this point the two changes interact. For instance, altering Telephone Number Formatting to format the number in a different way may require adapting Telephone Number Validating.

We saw that the dependencies between changes could be as complex as feature dependencies in feature modeling and accordingly represented by feature diagrams. For dependencies appearing among features without a common parent, additional constraints expressed as logical expressions [27] could be used. These constraints can be partly embedded into feature diagrams by allowing them to be directed acyclic graphs instead of just trees [10].

Some dependencies between changes may exhibit only recommending character, i.e. whether they are expected to be included or not included together, but their application remains meaningful either way. An example of this are features that belong to the same change request. Again, feature modeling can be used to model such dependencies with so-called default dependency rules that may also be represented by logical expressions [27].

6. Evaluation and Tool Support Outlooks

We have successfully applied the aspect-oriented approach to change realization to introduce changes into YonBan, the student project management system discussed in previous section. YonBan is based on J2EE, Spring, Hibernate, and Acegi frameworks. The YonBan architecture is based on the Inversion of Control principle and Model-View-Controller pattern.

We implemented all the changes listed in Sect. 5.3. No original code of the system had to be modified. Except in the case of project registration statistics and project registration constraint, which where well separated from the rest of the code, other changes would require extensive code modifications if they have had been implemented the conventional way.

As we discussed in Sect 5.4, we encountered one change interaction: between the telephone number formatting and validating. These two changes are interrelated — they would probably be part of one change request — so it comes as no surprise they affect the same method. However, no intervention was needed in the actual implementation.

We managed to implement the changes easily even without a dedicated tool, but to cope with a large number of changes, such a tool may become crucial. Even general aspect-oriented programming support tools — usually integrated with development environments — may be of some help in this. AJDT (AspectJ Development

Tools) for Eclipse is a prominent example of such a tool. AJDT shows whether a particular code is affected by advices, the list of join points affected by each advice, and the order of advice execution, which all are important to track when multiple changes affect the same code. Advices that do not affect any join point are reported in compilation warnings, which may help detect pointcuts invalidated by direct modifications of the application base code such as identifier name changes or changes in method arguments.

A dedicated tool could provide a much more sophisticated support. A change implementation can consist of several aspects, classes, and interfaces, commonly denoted as types. The tool should keep a track of all the parts of a change. Some types may be shared among changes, so the tool should enable simple inclusion and exclusion of changes. This is related to change interaction, which can be addressed by feature modeling as we described in the previous section.

7. Related Work

The work presented in this paper is based on our initial efforts related to aspect-oriented change control [8] in which we related our approach to change-based approaches in version control. We concluded that the problem with change-based approaches that could be solved by aspect-oriented programming is the lack of programming language awareness in change realizations.

In our work on the evolution of web applications based on aspect-oriented design patterns and pattern-like forms [1], we reported the fundamentals of aspect-oriented change realizations based on the two level model of domain specific and generally applicable change types, as well as four particular change types: Class Exchange, Performing Action After Event, and One/Two Way Integration.

Applying feature modeling to maintain change dependencies (see Sect. 4) is similar to constraints and preferences proposed in SIO software configuration management system [4]. However, a version model for aspect dependency management [23] with appropriate aspect model that enables to control aspect recursion and stratification [2] would be needed as well.

We tend to regard changes as concerns, which is similar to the approach of facilitating configurability by separation of concerns in the source code [9]. This approach actually enables a kind of aspect-oriented programming on top of a versioning system. Parts of the code that belong to one concern need to be marked manually in the code. This enables to easily plug in or out concerns. However, the major drawback, besides having to manually mark the parts of concerns, is that — unlike in aspect-oriented programming — concerns remain tangled in code.

Others have explored several issues generally related to our work, but none of these works aims at actual capturing changes by aspects. These issues include database schema evolution with aspects [12] or aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions [3]. Also, an increased changeability of components implemented using aspect-oriented programming [17], [18], [22] and aspect-oriented programming with the frame technology [19], as well as enhanced reusability and evolvability of design patterns achieved by using generic aspect-oriented languages to implement them [24] have been reported. The impact of changes implemented by aspects has been studied using slicing in concern graphs [15].

While we do see potential of aspect-orientation for configuration and reconfiguration of applications, our current work does not aim at automatic adaptation in application evolution, such as event triggered evolutionary actions [21], evolution based on active rules [6], adaptation of languages instead of software systems [16], or as an alternative to version model based context-awareness [7], [13].

8. Conclusions and Further Work

In this paper, we have described our approach to change realization using aspect-oriented pro-

gramming and proposed a feature modeling based approach of dealing with change interaction. We deal with changes at two levels distinguishing between domain specific and generally applicable change types. We described change types specific to web application domain along with corresponding generally applicable changes. We also discussed consequences of having to implement a change of a change.

The approach does not require exclusiveness in its application: a part of the changes can be realized in a traditional way. In fact, the approach is not appropriate for realization of all changes, and some of them can't be realized by it at all. This is due to a technical limitation given by the capabilities of the underlying aspect-oriented language or framework. Although some work towards addressing method-level constructs such as loops has been reported [14], this is still uncommon practice. What is more important is that relying on the inner details of methods could easily compromise the portability of changes across the versions since the stability of method bodies between versions is questionable.

Change interaction can, of course, be analyzed in code, but it would be very beneficial to deal with it already during modeling. We showed that feature modeling can successfully be applied whereby change realizations would be modeled as variable features of the application concept. Based on such a model, change dependencies could be tracked through feature dependencies. In the absence of a feature model of the application under change, which is often the case, a partial feature model can be developed at far less cost to serve the same purpose.

For further evaluation, it would be interesting to develop catalogs of domain specific change types of other domains like service-oriented architecture for which we have a suitable application developed in Java available [25]. Although the evaluation of the approach has shown the approach can be applied even without a dedicated tool support, we believe that tool support is important in dealing with change interaction, especially if their number is high.

By applying the multi-paradigm design with feature modeling [27] to select the generally applicable changes (understood as paradigms) appropriate to given application specific changes we may avoid the need for catalogs of domain specific change types or we can even use it to develop them. This constitutes the main course of our further research.

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

References

[1] M. Bebjak, V. Vranić, and P. Dolog. Evolution of web applications with aspect-oriented design patterns. In M. Brambilla and E. Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, pages 80–86, Como, Italy, July 2007.

[2] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In R. Hirschfeld et al., editors, *Proc. of NODe 2006*, LNI P-88, pages 49–64, Erfurt, Germany, Sept. 2006. GI.

[3] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. Reliable, secure, and transacted web service compositions with AO4BPEL. In *4th IEEE European Conf. on Web Services (ECOWS 2006)*, pages 23–34, Zürich, Switzerland, Dec. 2006. IEEE Computer Society.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programing: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[6] F. Daniel, M. Matera, and G. Pozzi. Combining conceptual modeling and active rules for the design of adaptive web applications. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[7] F. Dantas, T. Batista, N. Cacho, and A. Garcia. Towards aspect-oriented programming for context-aware systems: A comparative study. In *Proc. of 1st International Workshop on Software Engineering for Pervasive Computing Ap-*

plications, Systems, and Environments, SEPCASE'07, Minneapolis, USA, May 2007. IEEE.

[8] P. Dolog, V. Vranić, and M. Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, Dec. 2001.

[9] Z. Fazekas. Facilitating configurability by separation of concerns in the source code. *Journal of Computing and Information Technology (CIT)*, 13(3):195–210, Sept. 2005.

[10] R. Filkorn and P. Návrat. An approach for integrating analysis patterns and feature diagrams into model driven architecture. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *Proc. 31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, LNCS 3381, Liptovský Jan, Slovakia, Jan. 2005. Springer.

[11] S. Goldschmidt, S. Junghagen, and U. Harris. *Strategic Affiliate Marketing*. Edward Elgar Publishing, 2003.

[12] R. Green and A. Rashid. An aspect-oriented framework for schema evolution in object-oriented databases. In *Proc. of the Workshop on Aspects, Components and Patterns for Infrastructure Software (in conjunction with AOSD 2002)*, Enschede, Netherlands, Apr. 2002.

[13] M. Grossniklaus and M. C. Norrie. An object-oriented version model for context-aware data management. In M. Weske, M.-S. Hacid, and C. Godart, editors, *Proc. of 8th International Conference on Web Information Systems Engineering, WISE 2007*, LNCS 4831, Nancy, France, Dec. 2007. Springer.

[14] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proc. of 5th International Conference on Aspect-Oriented Software Development, AOSD 2006*, pages 63–74, Bonn, Germany, 2006. ACM.

[15] S. Khan and A. Rashid. Analysing requirements dependencies and change impact using concern slicing. In *Proc. of Aspects, Dependencies, and Interactions Workshop (affiliated to ECOOP 2008)*, Nantes, France, July 2006.

[16] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Functional approach to the adaptation of languages instead of software systems. *Computer Science and Information Systems Journal (ComSIS)*, 4(2), Dec. 2007.

[17] A. A. Kvale, J. Li, and R. Conradi. A case study on building COTS-based system using aspect-oriented programming. In *2005 ACM Symposium on Applied Computing*, pages 1491–1497, Santa Fe, New Mexico, USA, 2005. ACM.

[18] J. Li, A. A. Kvale, and R. Conradi. A case study on improving changeability of COTS-based system using aspect-oriented programming. *Journal of Information Science and Engineering*, 22(2):375–390, Mar. 2006.

[19] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects. In *Workshop on Aspects, Componentsand Patterns for Infrastructure Software (held with AOSD 2004, International Conference on Aspect-Oriented Software Development)*, Lancaster, UK, Mar. 2004.

[20] R. Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[21] F. Molina-Ortiz, N. Medina-Medina, and L. García-Cabrera. An author tool based on SEM-HP for the creation and evolution of adaptive hypermedia systems. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[22] O. Papapetrou and G. A. Papadopoulos. Aspect-oriented programming for a component based real life application: A case study. In *2004 ACM Symposium on Applied Computing*, pages 1554–1558, Nicosia, Cyprus, 2004. ACM.

[23] E. Pulvermüller, A. Speck, and J. O. Coplien. A version model for aspect dependency management. In *Proc. of 3rd Int. Conf. on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 70–79, Erfurt, Germany, Sept. 2001. Springer.

[24] T. Rho and G. Kniesel. *Independent evolution of design patterns and application logic with generic aspects — a case study.* Technical Report IAI-TR-2006-4, University of Bonn, Bonn, Germany, Apr. 2006.

[25] V. Rozinajová, M. Braun, P. Návrat, and M. Bieliková. Bridging the gap between service-oriented and object-oriented approach in information systems development. In D. Avison, G. M. Kasper, B. Pernici, I. Ramos, and D. Roode, editors, *Proc. of IFIP 20th World Computer Congress, TC 8, Information Systems*, Milano, Italy, Sept. 2008. Springer Boston.

[26] V. Vranić. Reconciling feature modeling: A feature modeling metamodel. In M. Weske and P. Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS

3263, pages 122–137, Erfurt, Germany, Sept. 2004. Springer.

[27] V. Vranić. Multi-paradigm design with feature modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2(1):79–102, June 2005.

[28] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing applications with aspect-oriented change realization. In *Proc. of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, LNCS, Brno, Czech Republic, 2008.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Two Hemisphere Model Driven Approach for Generation of UML Class Diagram in the Context of MDA

Oksana Nikiforova*

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

oksana.nikiforova@rtu.lv

Abstract

The Model Driven Architecture (MDA) separates the system business aspects from the system implementation aspects on a specific technology platform. MDA proposes a software development process in which the key notions are models and model transformation, where the input models are platform independent and the output models are platform specific and can be transformed into a format that is executable. In this paper principles of MDA and model transformations are applied for generation of UML class diagram from two hemisphere model, which is presented in the form of business process model related with concept model. Two hemisphere model is developed for the problem domain concerned with an application for driving school and UML class diagram is generated using the approach offered in the paper.

1. Introduction

One of the modern research goals in software engineering is to find a software development process, which would provide fast and qualitative software development. Most of currently proposed methodologies and approaches try to make the development process easier and still more qualitative. For achievement of this goal the role of explicit models becomes more and more important. Lately, the most popular approach is Model Driven Architecture [18]. MDA is the central component in the OMG's strategy for maximizing return on investment, reducing development complexity and future-proofing against technological change [29]. MDA tools do not support the complete code-generation capabilities from the initial business information, and the most problematic stage is system modelling based on knowledge about problem domain [22].

The main idea of MDA is to achieve formal system representation at the highest level of abstraction. Nowadays MDA tools support translation of platform independent system presentation into software components and code generation and researchers try to "raise" it as high as possible to fulfill the main statement of the MDA [13]. One of the most important and problematic stages in MDA realization is derivation of PIM elements from a problem domain and PIM construction in the form that is suitable for the PSM. It is necessary to find the way to develop PIM using formal representation, so far keeping the level of abstraction high enough. PIM model should represent system static and dynamic aspects. Class diagram shows static structure of the developed system. But UML is a modelling language and does not have all the possibilities to specify context and the way of modelling, which is always required to be defined in a methodology. Therefore, the construction of class diagram has to be based on well defined rules for its elements generation from the problem domain model presented in the suitable form.

Class diagram discussed in the paper contains classes, relations among them, attributes and operations of classes. Dynamic aspects, which are another meaningful component of system presentation at the platform independent level is not the object of the current research. To obtain the class diagram the initial business knowledge represented with two hemisphere model may be used. The transformation of this model into class diagram is discussed in the paper. The transformation should be defined in formal way and should be acceptable for use in transformation tool. The structure of a transformation tool is discussed in [13] with definition of models, necessary for transformation and transition between these models. Transformation tools take a source model as an input, and create another model, called target model, as an output [13]. Therefore, implementation of transformation needs well-defined set of notational elements of source and target models and definition for transformation of elements of one model into elements of another one. The paper describes class diagram development based on two hemisphere model. Therefore, according to Kleppe's definition source model is defined in terms of two hemisphere model (business process and concept model) and target model is defined in terms of UML class diagram [28]. The structure of the paper is as follows. The next section presents main principles of model driven architecture, defines models to be developed within it, describes transformations to be formalized to be able to develop the tool for support of that transformations. Section 3 presents an information about using of two hemisphere model to fulfil the main statement of MDA corresponds to formal transformations between models. An essence of two hemisphere model is shown in several aspects of its historical evolution and refinement and transformations of two hemisphere model into elements of class diagram are described according to the present state of author's investigations. The transformations presented in the paper are verified in section 4, where the approach offered in the paper is applied for several problem domains. Due to limitations on volume of the paper the section shows only general results on these applications. Section 5 concludes on the research presented in the paper and gives several remarks on author's future work in the area of model transformations.

2. Main Principles of Model Driven Architecture

MDA introduces an approach to system specification that separates the views on three different layers of abstraction: high level specification of how system is working (Computation Independent Model or CIM), the specification of system functionality, i.e. of what the system is expected to do (Platform Independent Model or PIM) and the specification of the implementation of that functionality on a specific technology platform (Platform Specific Model or PSM). In OMG Model Driven Architecture these models are primary artefacts in software developments process and all the activities are concentrated on going from CIM to PIM, from PIM to PSM and from PSM to code. The very important role there is played by the quality of PIM, i.e. its capability to adequately represent system under development [18].

2.1. Models within the MDA

CIM presents the requirements for the system to be modelled in a platform independent model, describing the situation in which the system will be used. Such a model is sometimes called a domain model or a business model. It may hide much or all information about the use of automated data processing systems. A CIM is a model of a system that shows the system in the environment in which it will operate, and thus it helps in presenting exactly what the system is expected to do. It is useful, not only as an aid to understanding a problem, but also as a source of a shared vocabulary for use in other models [18]. PIM is describing that part of information system specification, which is close to code, but is independent of platform specific features. PIM is representing information system in that way that will remain un-

changed on any programming platform. Nevertheless PIM usually is accommodated to specific architecture style [18]. Platform Independent Model is the model that resolves business requirements through purely problem-space terms and it does not include platform specific concepts. The PIM provides formal specification of the structure and functionality of the system that abstracts away technical details. There has to be rules for PIM checking if it defines all problem domain concepts in the correct way [18]. Platform Specific Model is a solution model that resolves both functional and non-functional requirements through the use of platform specific concepts. The platform definition can include wide range of conceptions in the context of MDA. It can be operation system, programming language, any technological platform, such as CORBA, Java 2 Enterprise Edition, also any specific vendor platform (for example, Microsoft .NET) [18]. Platform can imply any of engineering and technological characteristics, which are not important for program unit fundamental business functionality [18].

2.2. Model Transformations within the MDA

Generally, system model refinement and evolution in the framework of MDA is presented in Figure 1.

CIM presents specification of the system at problem domain level and can be transformed into initial elements of PIM. PIM provides formal specification of the system structure and functions that abstracts from technical details, and thus presents solution aspects of the system to be developed. Development of the solution domain model is based on derivation of all the necessary elements from problem domain description (Transformation 1 in Fig. 1). The PIM received as a result of Transformation 1 has to be refined (Transformation 2 in Fig. 1) to get a form suitable for PSM generation, i.e. PIM-refined enables model transformation (Transformation 3 in Fig. 1) to the platform level, named Software Domain in Figure 1.

An MDA idea is promising – raising up the level of abstraction, on which systems are developed, we could develop more complex systems more qualitatively. Core of solution domain development strategies focuses on the transformation of system model from the aspects of business level into the application level (Transformation 2 in Fig. 1). The main idea of MDA is to achieve formal system representation at the as high level of abstraction as possible. Nowadays MDA tools support translation of solution elements into software components (Transformation 3 in Fig. 1) and code generation (Transformation 4 in Fig. 1), and researchers try to "raise" it up as high as possible to fulfil the main statement of the MDA [18].

Transformations 1 and Transformation 2 are defined within different solutions [33], [36], [16], [30], [12], but there is no any solution, where complete transformation CIM $\rightarrow$ PIMinitial $\rightarrow$ PIMrefined would be defined [22].

One of the most important and problematic stages in MDA realization is derivation of PIM elements from a problem domain, and PIM construction in the form that is suitable for the PSM. Solutions that are focused on Transformation 1 can't insure that a PIM contains all the necessary information, and that the presentation of the PIM is formal enough to be able to transform it into the correct PSM, that is to support already the Transformation 3 [22]. It is necessary to find the way to develop PIM using formal representation, so far keeping the level of abstraction high enough, i.e. to implement Transformation 2 in formal way. The central element of PIM is the presentation of system structure, which would be independent from further implementation and usually is presented in the form of class diagram in UML notation [28], as well as adequate presentation of system dynamics. Different modelling tools are used for that. The paper discusses the class diagram development aspects, which satisfies the main statement of MDA and are based on transformation from two hemisphere model into elements of class diagram defined in UML.

Currently, transformations between UML models are still a subject of intensive investiga-

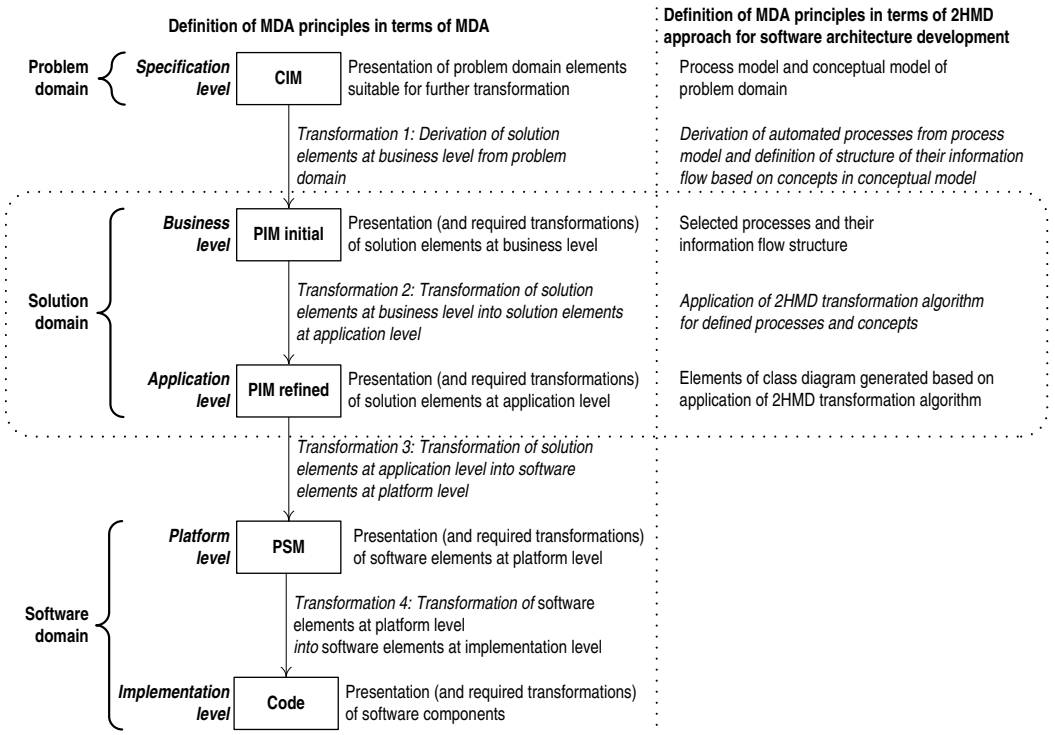


Figure 1. General structure of model transformation in the framework of MDA

tion. Principles of simple language for transformations are presented in [13]. Several proposals [6] are made in response to OMG request for proposals to MOF Query/View/Transformation [6]. The great attention is devoted to UML class diagram development, because class diagram in UML-based CASE systems serves as a main source of knowledge for development of software system: database specification, graphical user interface, application code, etc. [35].

Class diagram is the most often used model for visual representation of static aspects of classes [35]. Class diagrams in object-oriented software development are typically used: as domain models to explore domain concepts; as conceptual/analysis models to analyse requirements; as systems design models to depict detailed design of object-oriented software [1]. But UML is a modelling language and does not have all the possibilities to specify context and the way of modelling, which is required always to be defined in a methodology. There fore the construction of class diagram has to be based on well defined rules for its elements generation from the problem domain model presented in the form suitable for that.

2.3. Structure of a Tool for Model Transformation

The MDA process shows the role that the various models, PIM, PSM, and code play within the MDA framework. A transformation tool takes a PIM and transforms it into a PSM. A second (or the same) transformation tool transforms the PSM to code. These transformations are essential in the MDA development process. The transformation tool takes one model as input and produces a second model as its output. There is a distinction between the transformation itself, which is the process of generating a new model from another model, and the transformation definition. The transformation tool uses the same transformation definition for each transformation of any input model. A transformation is defined in [13] as the automatic generation of a target model from a source model, according to a transformation definition.

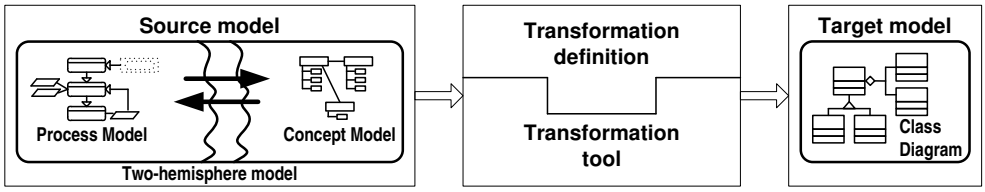


Figure 2. Schema of model transformation tool

And a transformation definition is defined in [13] as a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.

The recent tendency of automation of information handling process is essential in industry of information technology [9]. It gives a possibility to spare human and time resources. The implementation of tool, which automates transformation into class diagram, gives a possibility to receive static structure of the system without spending of a lot of time on design. For any transformation the initial data and needed result should be defined before. A transformation tool or approach takes a model on input, so called source model, and creates another model, so called target model, on output, see Fig. 2 [13]. The two hemisphere model has been marked as input with mapping rules, the class diagram and transformation trace has been received on output. Transformation trace shows the plan how an element of the two hemisphere model is transformed into the corresponding element of the class diagram, and which parts of the mapping are used for transformation of every part of the two hemisphere model [18]. Figure 2 shows how a transformation tool takes input – the two hemisphere model and receives output – the class diagram. Therefore implementation of model transformation (in our case transformation from two hemisphere model into class diagram) needs well-defined set of notational elements of source model, well-defined set of notational elements of target model and definition for transformation of elements of one model into elements of another one.

According to key notes of the paper the language for description of source model is defined as a notation for construction of two hemisphere model [21] and the language for description of target model is defined as a notation for construction of UML class diagram (see Fig. 2).

3. Models and Model Transformations in terms of Two Hemisphere Model

According to [32] the significant aspect of real world behaviour seen from the process point of view, where process is understood as the collection of actions, chronologically ordered and influencing objects and is more then "just an amorphous heap of the action". Similarly to the structural modelling of the real world [32]. Two hemisphere model corresponds to both fundamental things – functional aspects of the system defined in terms of business processes and the structural ones defined in terms of concept model. The details in the right column of the table in Figure 1 correspond to the two hemisphere approach, which addresses the construction of information about problem domain by use of two interrelated models at problem domain level, namely, the process model and the conceptual model. The conceptual model is used in parallel with process model to cross-examine software developers understanding of procedural and semantic aspects of problem domain.

3.1. Essence of Two Hemisphere Model

Two hemisphere model driven approach [21] proposes using of business process modelling and concept modelling to represent systems in the platform independent manner and describes how to transform business process models into UML models. For the first time the strategy was proposed in [20], where the general framework for object-oriented software development had been presented and the idea about usage of two interrelated models for software system develop-

ment has been stated and discussed. The strategy supports gradual model transformation from problem domain models into program components, where problem domain models reflect two fundamental things: system functioning (processes) and structure (concepts and their relations). The title of the proposed strategy [21] is derived from cognitive psychology [2]. Human brain consists of two hemispheres: one is responsible for logic and another one for concepts. Harmonic interrelated functioning of both hemispheres is a precondition of an adequate human behaviour. A metaphor of two hemispheres may be applied to software development process because this process is based on investigation of two fundamental things: business and application domain logic (processes) and business and application domain concepts and relations between them. Two hemisphere approach proposes to start process of software development based on two hemisphere problem domain model, where one model reflects functional (procedural) aspects of the business and software system, and another model reflects corresponding concept structures. The co-existence and inter-relatedness of these models enables use of knowledge transfer from one model to another, as well as utilization of particular knowledge completeness and consistency checks [21]. Figure 3 shows the essence of two hemisphere model for an example of an application for driving school.

A notation of the business process model, which reflects functional perspectives of the problem and application domains, is optional, however, it must reflect the following components of business processes: processes; performers; information flows; and information (data) stores [21]. For current research is used business process model constructed with GRAPES [11] notation. Current functional requirements always are present in the business process model that helps to maintain their consistency. As a result sophisticated models are used without disturbing software developers' and business experts' natural ways of thinking [21]. Some recent surveys show that about 80 percent of companies are engaged in business process improvement and redesign [10]. This implies that many companies are common with business process modelling techniques [10] or at least they employ particular business process description frameworks [31]. On the other hand practice of software development shows that functional requirements can be derived from problem domain task descriptions even about 7 times faster than if trying to elicit them directly from users [17]. Both facts mentioned above and existence of many commercial business modelling tools are a strong motivation to base software development on the business process model rather than on any other soft or hard models [21]. The concept model (graph G2 in Fig. 3) is used in parallel with business process model to cross-examine software developers understanding of problem and platform independent models. According to Larman [16] real-world classes with attributes relevant to the problem domain and their relationships are presented in concepts model. It is a variation of well known entity relationship (ER) diagram notation [4] and consists of concepts (i.e. entities or objects) and their attributes. Application of two hemisphere model for generation of class diagram gives a possibility to avoid relations between classes in concept model at business level (of problem domain). Due to transformation of process model into elements of object communication expressed in terms of UML communication diagram it becomes possible to define the relations between classes already according system realization at software level (of implementation domain). Therefore relations between concepts are not shown in concept model in Fig. 3). The notational conventions of the business process diagram gives a possibility to address concepts in concept model to information flows (e.g. events) in process model (see Fig. 3). All elements of the two-hemisphere model stated as source model in Figure 2 are as follows:

– Business process diagram/ Process – business process usually means a chain of tasks that produce a result which is valuable to some hypothetical customer. A business process is a gradually refined description of a business activity (task). Task is an atomic business process unit, which actually de-

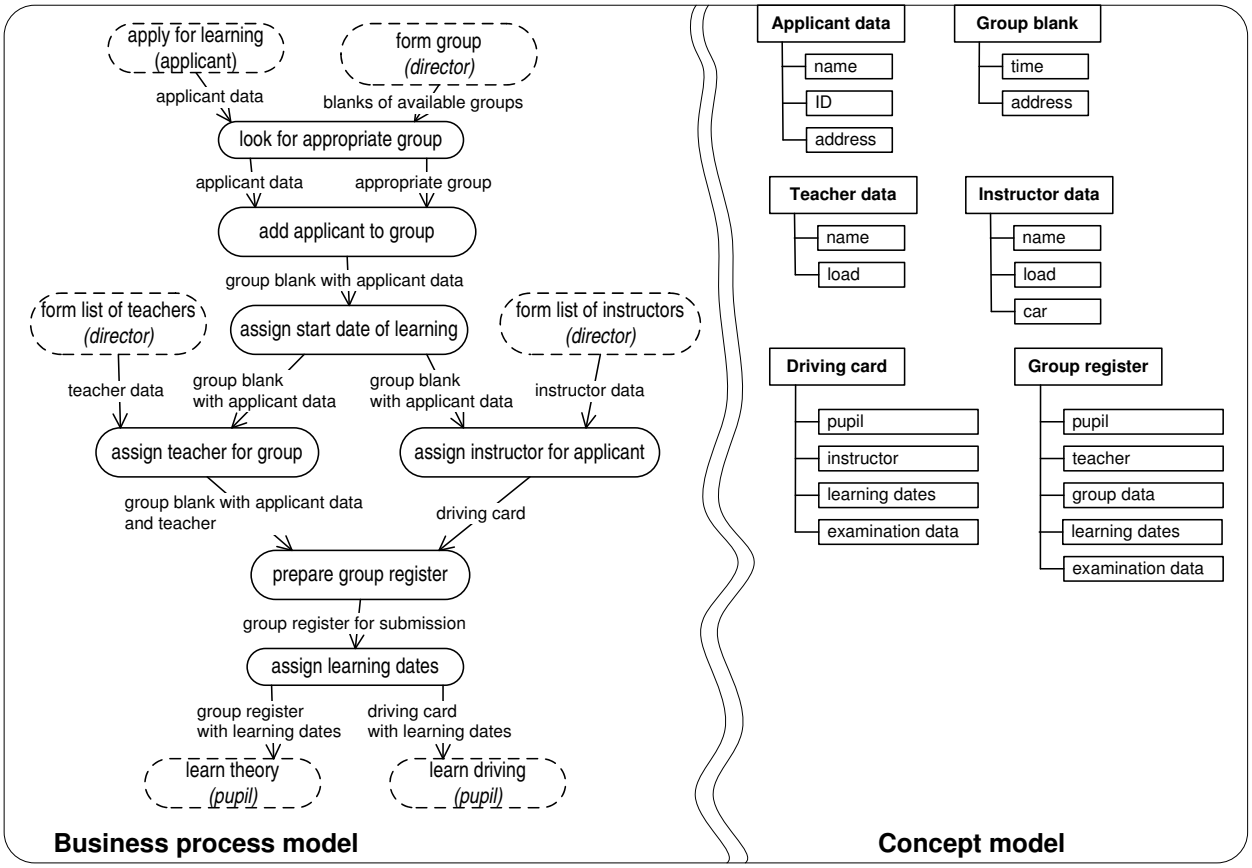


Figure 3. Example of two hemisphere model (application for driving school)

scribes some step or function and is done by a Performer [11].

- Business process diagram/Performer – performer is an attribute of a task of business process and serve as a resources required to perform the activities [11].
- Business process diagram/Event – events are an input/output object (or more precisely – the arrival of an input object and departure of an output object) of certain business process. These objects can be material things or just information [11].
- Concept model/Concept – conceptual classes that are software (analysis) class candidates in essence. A conceptual class is an idea, thing, or object. A conceptual class may be considered in terms of its symbols – words or images, intensions – definitions, and extensions – the set of examples [16].
- Concept model/Concept/Attribute – an attribute is a logical data value of an object [16].

The investigation of two hemisphere model driven approach under the MDA framework in [25] shows that approach could be applied for generation of several elements of class diagram. This paper shows the strategy of two hemisphere model application for generation of UML class diagram in a more precise way. The elements of the class diagram stated as target model in Figure 2 are as follows (only the main elements of the class diagram are listed here):

- Class diagram/Class – a class is the descriptor for a set of objects with similar structure, behaviour, and relationships [28].
- Class diagram/Actor – an actor specifies a role played by a user or any other system that interacts with the subject [28].
- Class diagram/Class/Attribute – an attribute is a logical data value of an object [28].
- Class diagram/Class/Operation – an operation is a specification of a transformation or

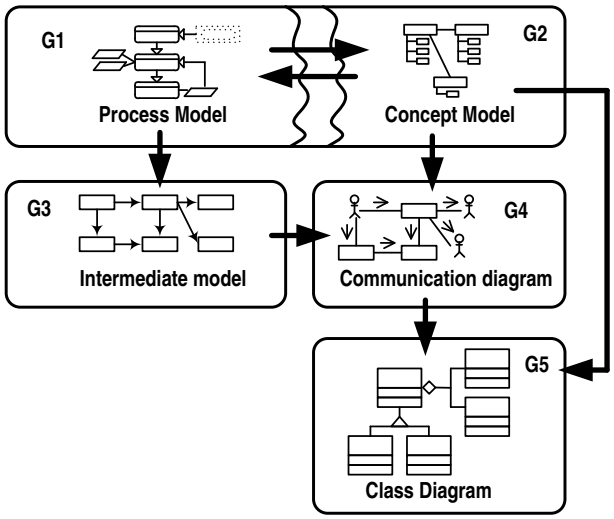


Figure 4. Transformations from two hemisphere model into class diagram under two hemisphere model driven approach

query that an object may be called to execute [28].

– Class diagram/Relationship – a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work [28].

It is necessary to find the way how source model elements can be transformed into target model elements according to the definition of transformations in the framework of MDA.

3.2. Description of Transformations between Models within Two Hemisphere Model Driven Approach

The two hemisphere model driven approach [20], [21], [27] proposes to apply transformations from business process model into scenarios for object interactions by using so called intermediate model, which is received in a direct transformation way from process model. Appropriate interacting objects are extracted from concept model. Class diagram is based on concept model and is formed according to information about object interaction. All defined transformations from two hemisphere model into elements of class diagram are shown in Figure 4. The schema presents the way how elements of business process model (graph G1 in Fig. 4) and concept model (graph G2 in Fig. 4) are transformed into elements of class diagram (graph G5 in Fig. 4), using intermediate model (graph G3 in Fig. 4) and UML communication diagram (graph G4 in Fig. 4) [25].

Analysis of two hemisphere model proposed in [22] and application of two hemisphere model for knowledge architecture development in the task of study program development presented in [22] makes to think that notational conventions of UML communication diagram is more suitable for definitions of formal transformations of two hemisphere model into object interaction and then into class diagram, than using of UML sequence diagram. Although the aspect of time sequence, which is a component of UML sequence diagram and is not shown in communication diagram, is missed in this case. And author of the paper now is investigating the possibility to save time aspect in transition from two hemisphere model into class diagram through the defined transformations [26].

Intermediate model (graph G3 in Fig. 4 and Fig. 5) is used to simplify the transition between business process model and model of object interaction, which is presented in the form of UML communication diagram (graph G4 in Fig. 4 and Fig. 5).

Intermediate model is a graph generated from business process models using methods of

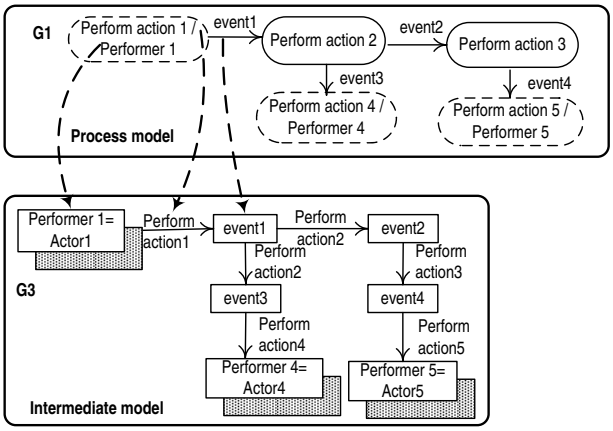


Figure 5. Transformations from business process model into intermediate model

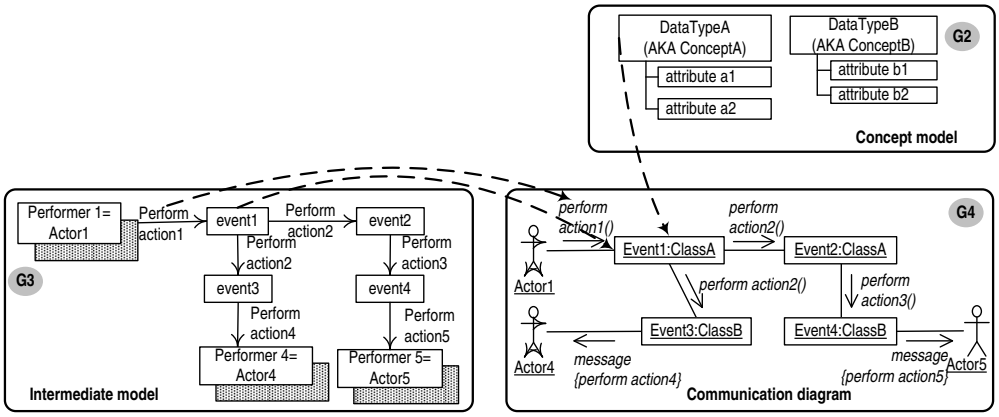


Figure 6. Transformations from intermediate model and concept model into object communication diagram

direct graph transformations based on principles of graph theory [8]. The nodes of the graph G1 in Figure 5 are transformed into the arcs of the graph G3 of Figure 5, and the arcs of the graph G1 in Figure 5 are transformed into the nodes of the graph G3 of Figure 5 [25].

In a case of abstract names of arcs and nodes of graphs in Figure 5 business process "perform action 1" is transformed into an arc "perform action 1" of intermediate model (graph G3 on Fig. 5) and events are transformed into nodes of intermediate model. Constructed intermediate model serves as a base for communication model. The communication diagram is represented as a graph G4 in Figure 4 and Figure 6.

The next transformation defines the method "perform action 1()" in communication diagram (graph G4 on Fig. 6) from the same arc of intermediate model, where the class-receiver of this method is defined as ClassA, because ConceptA defines a data type for event1 in concept model. Therefore if each process is examined as a message, and each data flow as an object, a draft communication diagram could be received by replacing all events of intermediate model with concerned class exemplars and the actions of intermediate model with messages or operations.

The last transformation of this business process defines the responsible class of this method in class diagram (graph G5 in Fig. 4 and Fig. 7) based on information that the type of the event "event 1" is defined by class in. The element "performer 1" is transformed as a node of intermediate model, and as "actor 1" of communication model. This element is defined as "actor 1" in class diagram. Data types for elements "event 1"

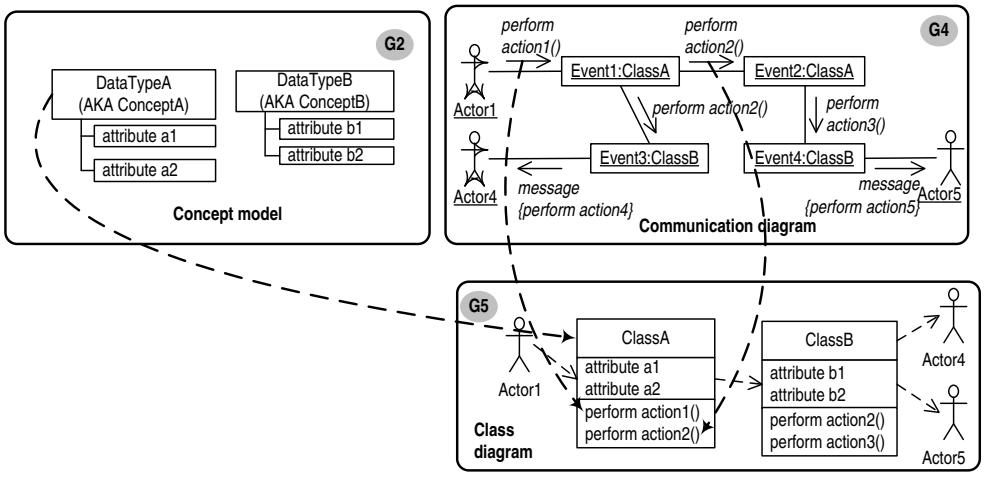


Figure 7. Transformations from intermediate model and object communication diagram into class diagram

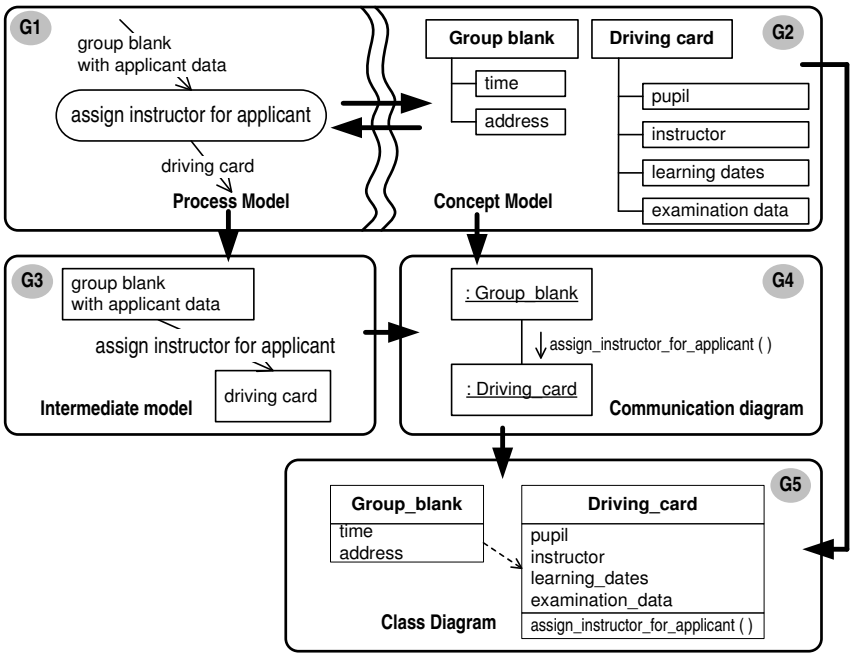


Figure 8. An example of transformation of process and concept elements into class elements

and "event 3" is defined as "DataType A" or "Concept A" of concept model.

These transformations are based on the hypothesis that elements of the class diagram (graph G5 in Fig. 7) can be received from the two hemisphere model by applying defined techniques of graph transformation [8]. The next step of transition is a class diagram. Here all messages of object communication (graph G4 in Fig. 7) are encapsulated as operations of classes using main principles of class diagram development based on information of object communication and all events and concepts defined as objects in graph G4 are defined as classes in class diagram (graph G5 in Fig. 7). Class diagram presents the set of attributes based on attributes defined in concept model.

In a very simple example, transformation described above looks like it is shown in Figure 8, where transformation of fragment of two hemisphere model for driving school into a fragment of the exact class is represented.

There is one process "assign instructor to applicant", which has an input event "group blank with applicant data", where concept "Group blank" with its attributes defines data type for the event, and an output event "driving card", where concept "Driving card" with its attributes defines data type for the event. These interrelated elements define a two hemisphere model, which serve as a base for transformation into intermediate model, with the same names of elements, but different position – arcs of process model are transformed into nodes of intermediate model, and nodes of business process are transformed into the arcs of intermediate model. Intermediate model allows to define communication diagram, where initial process "assign instructor to applicant" is defined as a method. And object-sender and object-receiver are defined in accordance with discussion above. Based on a communication of objects defined, it is possible to construct class diagram according to rules of object-oriented system modelling [20].

Experiments with different combinations of of incoming and outgoing arcs in the model of business process and a variety of different data types defined in a concept model, where the data type can be the same for incoming and outgoing events or different, give a possibility to define different types of relationships between classes. The results of full investigations of all the possible combinations of two hemisphere model, which gives a possibility to define relationships between classes, are shown in [24]. The paper has a discussion of a possibility to share class responsibilities and to encapsulate class attributes and methods according defined transformations from arcs and nodes of two hemisphere model. The paper offers the description of tool for the usage of two hemisphere model for class diagram generation based on the defined transformations.

4. Verification of Transformation within Two Hemisphere Model Driven Approach

When the structures of input and output data are known, it is possible to automate a process of input data transformation into output data.Class diagram generation should consist of four steps according to the application of two hemisphere model (see Fig. 9):

1. construction of two hemisphere model;
2. generation of model elements and their interrelations in some structured form;
3. application of the transformation rules defined (processing algorithm);
4. definition of class specification in well structured form suitable for class diagram construction (for example, XML format).

One of the tools for business process modelling, which gives a possibility to construct two interrelated models (business process and the concept ones), is GRADE [7]. Indeed, GRADE generates text descriptions of model with permanent structure, therefore it is chosen as a tool for development of two hemisphere model and further generation of textual files. It defines all the elements of the model and their relations from one into another. Generated text files serves as an input information into the tool developed and described in [27] in order to support the processing algorithm and XML file, which contains structure of the class diagram required. XML format of class specification gives a possibility to receive visual representation of class diagram in any tool, which supports import from XML for class diagram development. An ability to develop an automated tool for generation of class structure in XML format demonstrated in [27] proves, that transformations discussed in the paper are enough formal for programming. The tool is applied for generation of class diagram from two hemisphere model developed for problem domain of pupil application for learning in a driving school shown in Figure 3. Classes, attributes, operations and relations among classes, which could be determined from the business process diagram and the data structure, were defined applying discussed transformations from business process and concept model to class diagram. Figure 9 represents the structure of class diagram obtained.

One of the limitations of the approach is an impossibility to define the full specification of methods with its arguments. This could be one of the potential directions for future investiga-

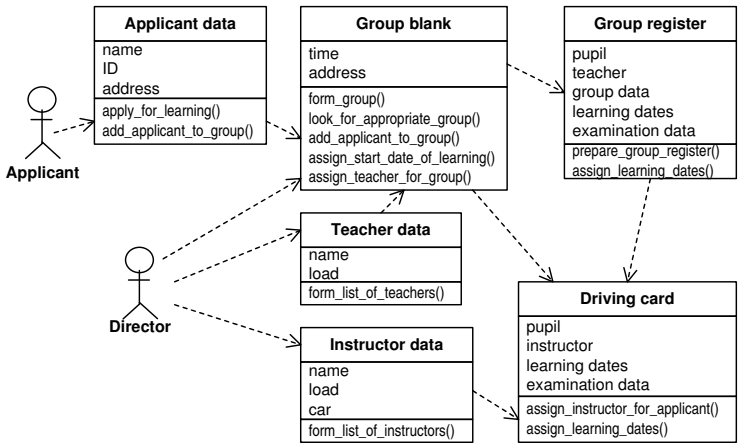


Figure 9. Initial structure of class diagram defined based on transformations from two hemisphere model

tion of application of two hemisphere model for generation of class diagram elements. In order to verify the transformations offered in the paper, the transformations defined for two hemisphere model driven approach in addition to an example of driving school are applied in some more examples: (1) problem area of hotel room reservation, where initial business process model is constructed in GRAPES notation [11] with CASE tool GRADE [7], the two hemisphere model is developed by authors and results of these experiments are shown in [23]; this case approve the possibility to define class diagram by application of the transformations offered in the paper; (2) problem area of insurance system, where initial model is constructed in GRAPES notation with CASE tool GRADE [7], the two hemisphere model is constructed by developers of GRADE tool as demo example and results of these experiments are shown in [25]; this case approve an independence of the transformations offered in the paper from the constructor of two hemisphere model. The problem area of hotel room reservation also is approbated by construction of initial business process model in IDEF0 [14] notation with CASE tool BPWin. Unfortunately, it does not provide construction of concept or object model. Therefore attributes of classes, received for hotel room reservation system with initial information in IDEF0 notation are missing in the class diagram. Even in this case automation of distributing methods among classes is important contribution within software development. Experiments on applying discussed transformations from two hemisphere model into class diagram in different problem domain prove that transformations are independent from problem domain. Experiments on applying transformation from two hemisphere model, constructed in various CASE tools and notations, prove that transformations from two hemisphere model into class diagram are independent from used notation of business process modelling, as well as CASE tool, used for initial model creation.

5. Conclusion

The Model Driven Architecture is the central component in the OMG's strategy for maximising return on investment, reducing development complexity and future-proofing against technological change [29]. But still the "complete code-generation capabilities" are no supported in MDA tools and the more problematic stage is exactly platform independent system modelling based on knowledge about problem domain. Since beginning of eighties a numerous accounts of model generated software systems have been offered to attack problems regarding software productivity and quality [3]. CASE tools developed up that time were oversold on their "complete code-generation capabilities" [15]. Nowadays, similar arguments are exposed to Object Management Group (OMG) Model Driven Architecture (MDA) [34], using and integrating Unified Mod-

elling Language (UML) models [28] at different levels of abstraction. Manipulation with models enables software development automation within CASE tools supported by MDA [5], [13], [19]. The paper discusses abilities on usage of problem domain knowledge presentation in terms of two hemisphere model, which contains two interrelated models of system aspects – process and concept presentation. The proposed transformations are applied to two hemisphere model of application for driving school and classes with attributes and different kinds of relationships are identified based on elements of process and concept models. The ability to define all the types of transformations in a formal way gives a possibility to automate the process of class diagram development from correct and precise two hemisphere model. On one hand, it enables knowledge representation in a form understandable for both business users and system analyst, moreover cover complete and consistent presentation of different system aspects. And on the another hand, it supports the formal transformations of model elements into elements of UML class diagram, which often is a starting point during software development by using nowadays CASE tools, especially in the ones following an idea of MDA. The central hypothesis of this research is that it is possible to apply the graph theory technique for model transformation in the framework of MDA, where the source model is defined in terms of a business process model, associated with a concept model, and the target model is defined in terms of a class diagram. Analysis of the system models presented as a set of graphs developed on the basis of the initial two hemisphere model enables derivation of the class diagram, which is the central component of PIM and is sufficiently detailed in order for the PSM to be generated. Two hemisphere model gives a possibility to define classes with attributes and operations they have to perform, as well as different types of relations can be defined between classes, based on analysis of different combinations of type definition for incoming and outgoing flows of processes of two hemisphere model. At the moment authors try to investigate the possibility of exact definitions of method's arguments based on information in two hemisphere model and to investigate abilities of usage of two hemisphere model for dynamic component of platform independent model expressed in terms of object interaction and state transition. A deeper analysis of notational conventions of nowadays available notations for business process modelling is required and could be stated as one of further researches directions.

Acknowledgements The research reflected in the paper is supported by the research grant No. FLPP-2009/10 of Riga Technical University "Development of Conceptual Model for Transition from Traditional Software Development into MDA-Oriented." And partially the research reflected in the paper is supported by Grant of Latvian Council of Science No. 09.1245 "Methods, models and tools for developing and governance of agile information systems".

References

[1] S. Ambler. *The Elements of UML Style*. Cambridge University Press, 2003.

[2] J. Anderson. *Cognitive Psychology and Its Implications*. W.H. Freeman and Company, New York, 1995.

[3] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, 1985.

[4] P. Chen. The entity relationship model – towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

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

[6] T. Gardner and C. A. Griffin. *Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations Towards the Final Standards*. Object Management Group. OMG documents ad/03-08-02, available at http://www.omg.org.

[7] GRADE Development Group. *GRADE tools*.

[8] J. Gross and J. Yellen. *Graph Theory and Its Applications*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2nd edition, 2006.

[9] M. Guttman and J. Parodi. *Real-Life MDA: Solving Business Problems with Model Driven Architecture*. San Francisco, CA: Morgan Kaufmann Publishers, 2007.

[10] P. Harmon. Business process management. In *Lecture Notes in Computer Science*, volume 5240/2008. Springer Berlin/Heidelberg, 2008.

[11] INFOLOGISTIK GmbH. *GRADE Business Modeling, Language Guide*, 1998.

[12] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 2002.

[13] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture – Practise and Promise*. Addison Wesley, 2003.

[14] Knowledge Based Systems Inc. *IDEF Integrated Definition Methods*. Available at http://www.idef.com/.

[15] J. Krogstie. Integrating enterprise and is development using a model driven approach. In *Proceedings of 13th International Conference on Information Systems Development-Advances in Theory, Practice and Education*, pages 43–53. Springer Science+Business media, New York, 2005.

[16] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, New Jersey, 2000.

[17] S. Lausen. Task descriptions as functional requirements. *IEEE Software*, 20:58–65, March/April 2003.

[18] *MDA Guide Version 1.0.1*, June 2003. Available at http://www.omg.org/docs/omg/03-06-01.pdf.

[19] S. Mellor and M. Balcer. *Executable UML. A Foundation for Model-Driven Architecture*. Addison-Wesley, Boston, 2002.

[20] O. Nikiforova. General framework for object-oriented software development process. *Scientific Proceedings of Riga Technical University*, 13:132–144, 2002.

[21] O. Nikiforova and M. Kirikova. Two-hemisphere driven approach: Engineering based software development. *Advanced Information Systems Engineering*, pages 219–233, June 2004.

[22] O. Nikiforova, M. Kirikova, and N. Pavlova. Principles of model driven architecture in knowledge modeling for the task of study program evaluation. *Databases and Information Systems IV*, pages 291–304, 2007.

[23] O. Nikiforova and N. Pavlova. Development of the tool for generation of uml class diagram from two-hemisphere model. *Proceedings of The third International Conference on Software Engineering Advances, International Workshop on Enterprise Information Systems*, pages 105–112, October 2008.

[24] O. Nikiforova and N. Pavlova. Foundations on generation of relationships between classes based on initial business knowledge. *Proceeding of the 17th International Conference on Information Systems Development, Information Systems Development: Towards a Service Provision Society*, August 2008. In press.

[25] O. Nikiforova and N. Pavlova. Open work of two-hemisphere model transformation definition into uml class diagram in the context of mda. *Preprint of the Proceedings of the 3rd IFIP TC 2 Central and East Europe Conference on Software Engineering Techniques, CEE-SET 2008*, pages 133–146, October 2008.

[26] O. Nikiforova and N. Pavlova. Modeling of object interaction with two-hemisphere model driven approach. 2009. Submitted to the 13th East-European Conference on Advances in Databases and Information Systems.

[27] O. Nikiforova, N. Pavlova, and J. Grigorjevs. Several facilities of class diagram generation from two-hemisphere model in the framework of MDA. *Proceedings of 23rd IEEE International Symposium on Computer and Information Sciences*, pages 1–6, 2008. Available at http://ieeexplore.ieee.org/.

[28] Object Management Group. *Unified Modeling Language Specification*. Available at http://www.omg.org.

[29] T. Pokorny. *The Model Driven Architecture: No Easy Answers*, 2005.

[30] T. Quatrany. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, second edition, 2000.

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

[32] V. Repa. Modelling business processes in public administration. *Advances in Information Systems Development. Bridging the Gap between Academia and Industry*, 1:107–118, 2006.

[33] J. Rumbaugh. Omt: The developing process. *Object Oriented Programming*, (8):14–18, 1995.

[34] J. Siegel. *Developing in OMG's Model-Driven Architecture*, 2001. OMG document omg/01-12-01. Available at http://www.omg.org/mda/papers.htm.

[35] T. Skersys and S. Gudas. Class model development using business rules. *Advances in Information Systems Development. Bridging the Gap between Academia and Industry*, 1:203–216, 2006.

[36] D. Tkach, W. Fang, and A. So. *Visual modeling technique: object technology using visual programming*. Addison Wesley, 1996.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Automated Code Generation from System Requirements in Natural Language

Jan Franců*, Petr Hnětynka*
*Faculty of Mathematics and Physics, Department of Software Engineering, Charles University in Prague
jfrancu@gmail.com, hnetynka@dsrg.mff.cuni.cz

Abstract

An initial stage of a software development is specification of the system requirements. Typically, these requirements are expressed in UML and consist of use cases and domain model. A use case is a sequence of tasks, which have to be performed to achieve a specific goal. The tasks of the use case are written in a natural language. The domain model describes objects used in the use cases. In this paper, we present an approach that allows automated generation of executable code directly from the use cases written in a natural language. Usage of the generation significantly accelerates the system development, e.g. it makes immediate verification of requirements completeness possible and the generated code can be used as a starting point for the final implementation. A prototype implementation of the approach is also described in the paper.

1. Introduction

Development of software is covered by several stages from which one of the most important is the initial stage – collecting system requirements. These requirements can be captured in many forms, however, use of the Unified Modeling Language (UML) has become an industry standard at least for large and medium-size enterprise applications. Development with UML [8] is based on modeling the developed system at multiple levels of abstraction. Such a separation helps developers to reflect specific aspects of the designed system on different levels and therefore to get a "whole picture" of the system.

Development with the UML starts with definition of goals of the system. Then, main characteristics of the system requirements are identified and described. A behaviour of the developed system is specified as a set of *use cases*. A *use case* is a description of a single task performed in the designed system [3]. The task itself is further divided into a sequence of steps that are performed by communicating entities. These entities are either parts of the system or users of the system. A step of a use case is specified by natural language sentences. The use cases of the system are completed by a *domain model* that describes entities, which together form the designed system and which are referred to in the use cases.

Bringing a system from the design stage to the market is a very time-consuming and also money-consuming task. A possibility to generate an implementation draft directly from the system requirements would be very helpful for both requirement engineers and developers and it would significantly speed up development of the system and decrease time required to deliver the system to the market and also decrease amount of money spent. The system use cases contain work-flow information and together with the domain model capture all important information and therefore seem to be sufficient for such a generation. But the problem is that the use cases are written in a natural language and there is a gap to overcome to generate the system code.

1.1. Goals of the paper

In this paper, we describe an approach, which allows to generate an implementation of a system from the use cases written in a natural language. The process proposed in the paper enables software developers to take an advantage of the carefully written system requirements in order to accelerate the development and to provide immediate feedback for the project's requirement engineers by highlighting missing parts of the system requirements. The process fits in the incremental development process where in each iteration developers can eliminate shortcomings in design. In addition, the process can be customized to fit in any enterprise application project.

Described approach is implemented in a proof-of-the-concept tool and tested on a case study.

To achieve the goals, the paper is structured as follows. Section 2 provides an overview of the UML models and technologies required for use case analysis. Section 3 shows how our generation tool is employed in the application development process. Section 4 describes the tool and all generation steps in detail while Section 5 presents particular examples of the generated code. Section 6 evaluates our approach and the paper is concluded in Section 7, where future plans are also shown.

2. Specification of Requirements

The *Unified Modeling Language* (UML) is a standardized specification language for the software development. Development with UML is based on modeling a system at multiple levels of abstraction in separated models. Each model represented as a set of documents clarifies the abstraction on a particular level and captures different aspects of the modeled system. The UML-based methodologies standardize whole development process and ensure that the designed system will meet all the requirements. UML also increases possibilities to reuse existing models and simplifies reuse of code. The developers can use several existing modeling tools/frameworks (e.g. [10]) to support this process.

In this paper, we work with the UML documents created during the initial stage of the system development, i.e. with the requirement specification. Results of the stage are captured in *use cases* and *domain model*.

2.1. Use Cases

A *use case* in the context of UML is a description of a process where a set of entities cooperates together to achieve a goal of the use case. The entities in the use case can refer to the whole system, parts of the system, or users. Each use case has a single entity called *system under discussion* (*SuD*); from the perspective of this entity, the whole use case is written. An entity primarily communicating with *SuD* is called a *primary actor* (*PA*). Other entities involved in the use case are called *supporting actors* (*SA*).

Each use case is a textual document written in a natural language. The book [3] recommends the following structure of the use case: (1) header, (2) main scenario, (3) extensions, and (4) sub-variations.

The header contains the name of the use case, *SuD* entity, primary actor and supporting actors. The main scenario (also called the *success scenario*) defines a list of steps (also called *actions*) written as sentences in a natural language that are performed to achieve the goal of the use case. An action can be extended with a *branch action*, which reflects possible diversions from the main scenario. There are two types of the branch actions: *extensions* and *sub-variations*. In an *extension*, actions are performed in addition to the extended action, while in a *sub-variation*, actions are performed instead of the extended action. The first sub-action of a branch action is called a *conditional label* and describes necessary condition under which the branch action is performed.

The above described structure is not the only possible one – designers can use any structure they like. In our approach, we assume the use cases satisfy these recommendation as it allows

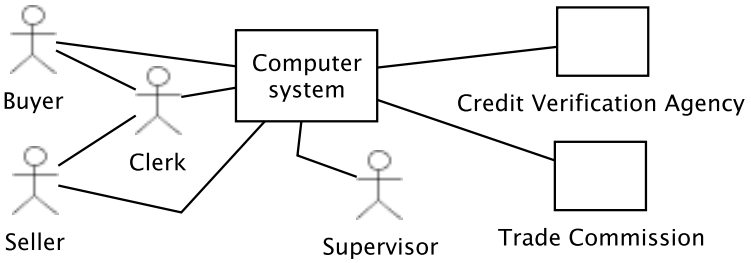


Figure 1. The Marketplace project entities

UseCase: Buyer buys a selected item
SuD: Clerk
PA: Buyer
Supporting actor: Computer System

Main success scenario specification:
1 Buyer submits to the clerk a reference to a selected offer.
2 Clerk submits the reference to the system.
3 Clerk reports the system response to the seller and requests billing and shipping information, payment method and payment details.
4 Buyer submits to the clerk the requested billing and shipping information, payment method and payment details.
5 Clerk enters the billing and shipping information, payment method and payment details.
6 Clerk reports the system response (with the unique acknowledgment) to the buyer.

Extensions:
3a System failed to validate the offer.
3a1 Use case abort.

Figure 2. Use case example

us to process the use case automatically and generate the system implementation. Such an assumption does not limit the whole approach in a significant way, hence the book [3] is widely considered as a "bible" for writing the use cases (in addition, we already have an approach for using use cases in fact with any structure – see Sect. 7).

In the rest of the paper we use as an example a *Marketplace* project for on-line selling and buying. A global view of the application entities is depicted on Figure 1. There are several actors, which communicate with the system. *Seller*s enter offers to the system and *Buyer*s search for interesting offers. Both of them mainly communicate directly with the *Computer system* – in few cases, they have to communicate through a *Clerk* who passes information to the Computer system. There is also a *Supervisor* which maintains the Computer system. A *Credit verification agency* verifies Seller's and Buyer's operations and finally a *Trade commission* confirms the offers.

The use case on Figure 2 is a part of the Marketplace specification (the whole specification has 19 use cases) and it describes communication between the Buyer (as PA), Clerk (as SuD), and Computer system. It is prepared according to the recommendations.

2.2. Domain Model

A *domain model* describes entities appearing in the designed system. Typically, the domain model is captured as a UML class diagram and consists of three types of elements: (1) *conceptual classes*, (2) *attributes* of conceptual classes, and (3) *associations* among conceptual classes.

Conceptual classes represent objects used in the system use cases. The attributes are features of the represented objects and associations describe relations among the classes. Figure 3 shows the Marketplace domain model.

As described in [8], noun phrases appearing in the use cases can be used for determining class names during creation of the domain model (in further detail, such a relation between the class diagrams and use cases is investigated in [1]).

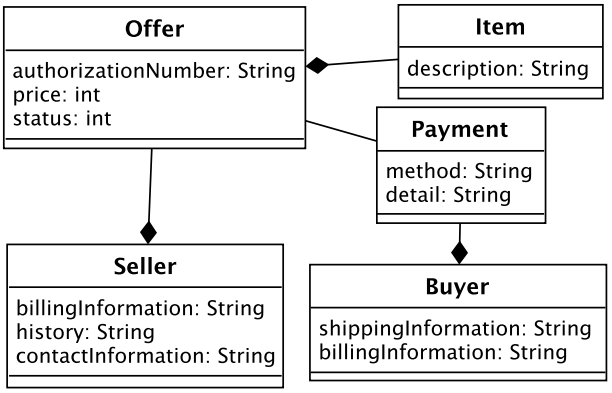


Figure 3. Marketplace domain model

2.3. Procasor Tool and Procases

The *Procasor* [6] is a tool for automated transformation of natural language (English) use cases into a formal behaviour specification. The transformations are described in [9] and further extended in [4] where almost all restrictions of a use case step syntax were removed.

As a formalism into which the natural language use cases are transformed the Procasor uses *procases* [9] that are a special form of *behaviour protocols* [14]. In addition to procases, a UML state machine diagram is also generated.

A procase is a regular expression-like specification, which can describe behaviour of a single entity as well as of the whole system [13]. The procases generate so called *traces* that represent all possible valid sequences of actions described by the use cases. Figure 7 shows a procase derived from the use case shown in Figure 2.

A procase is composed of operators (i.e. *+*, *;*), procedure calls (*{*,*}*), action tokens, and supporting symbols (i.e. round parenthesis for specifying operators' precedence). Each action token represents a single action that has to be performed and its notation is composed of several parts. First, there is a single character representing a type of the action. The possible types are *?* resp. *!* for request receiving resp. sending action, *#* for internal actions (unobservable by others than *SuD*) and *%* for special actions. The action type is followed by the entity name on which the action is performed. Finally, the name of the action itself is the last part (separated by a dot). In a case, there is no entity name, the action is internal. For example, *?B.submitSelectOffer* is the *submitSelectOffer* action where SuD waits for a request from the *B* (Buyer) entity.

The procases use the same set of operation as regular expressions. These are: *** for iteration, *;* for sequencing, and *+* for alternatives. In this paper, we call the *alternative* operator as a *branch action*, its operands (actions) as *branches*, and the *iteration* operator with its operand as a *loop action*.

A special action is *NULL* which means no activity and is used in places with no activity but the procase syntax requires an action specified there (e.g. with the alternative operator). Another special action is the first action inside a non-main scenario branch, which is alled *condition branch label* and expresses the condition under which the branch is triggered. Finally, the *%ABORT* special action represents a failure ending of the procase.

Procedure calls (written as a sequence of actions in curly brackets) represent a behaviour (mostly composed of inner actions) of the request receive action after which they are placed (the action is called *trigger action*).

2.4. Goals Revisited

As described in the sections above, the Procasor tool parses the use cases written in a natural language and generates a formal specification of behaviour of the designed system. A straightforward idea is then why to stop just with the generated behaviour description and not to generate also an implementation of the system which implements the work-flow captured in the use cases.

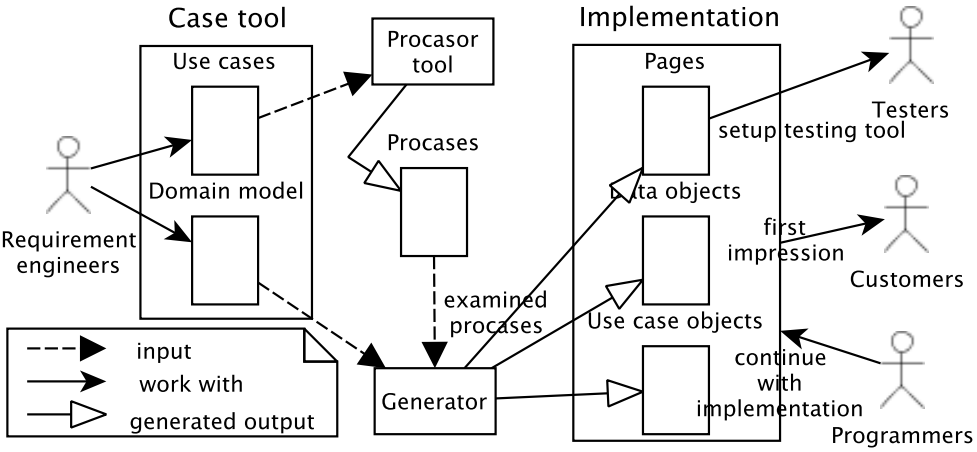


Figure 4. Development process overview

The goal of this paper is to present an extension of the Procasor tool that based on the use cases generates executable implementation of the designed system.

3. Generating Process

The development process with our generating tool is as follows. First, requirement engineers collect all requirements and describe them in the form of use cases. Then the Procasor tool automatically generates procases. In parallel, the requirement engineers create a project domain model. As a next step of validating the use cases, the generated procases can be reviewed. Then, our generator is employed and produces an implementation of the developed system. The generated implementation consists of three main parts: (i) use case objects where work-flow captured in a use case is generated, (ii) pages which are used to communicate with users of the system, and (iii) entity objects where the business logic is kept.

The generated implementation is only an initial draft and serves primarily for testing the use cases and domain model. But it can be also used as a skeleton for the actual implementation and/or to allow customers to gain first impressions of the application. The whole development process is illustrated in Figure 4.

At this point a common mistake has to be emphasized (which is also emphasized in [8]). The system requirements cannot be understood as final and unchangeable. Especially in incremental development, the requirements are created in several iterations and obviously the first versions are incomplete. Therefore if the generator is used on such input, it can generate a completely wrong implementation. But this implementation can be used to validate the use cases, repair them and regenerate the implementation.

4. Generating Tool in Detail

The generator of the implementation takes as an input the procases generated by the Procasor and the created domain model of the designed system. From these inputs, it generates the executable implementation.

The generation is automated and it consists of three steps:

1. First, procases generated from the Procasor are rearranged into a form, in which they still follow the procases syntax but are more suitable for generating the implementation (Sect. 4.1).
2. Then, a relation between words used in the use cases and elements in the domain model is obtained and parameters (i.e. their numbers and types) of the methods are identified (Sect. 4.2).
3. Finally, the implementation of the designed system is generated (Sect. 4.3).

4.1. Procase Preprocessing

The procases produced by the Procasor do not contain procedure calls brackets (see 2.3), which are crucial for successful transformation of the procases into the code. Except several marginal cases, each use case is a request-response sequence between SuD and PA (for enterprise applications). In the procase, a single request-response element is represented as a sequence of actions from which the first one is the request receive action (i.e. starts with ?) and then followed by zero or more other actions (i.e. sending request action, internal actions, etc.). In other words, SuD receives the request action, then performs a list of other actions, and finally returns the result (i.e. end of the initial request receive action). Hence, the sequence of actions after the request receive action can be modeled as a procedure content and enclosed in the procedure call brackets.

The following example is a simple procase in a form produced by the Procasor:

$$?PA.a; \#b; !SA.c; ?PA.d; \#e; \#f$$

After identifying the procedure calls, the procase is modified into the following form:

$$?PA.a\{\#b; !SA.c\}; ?PA.d\{\#e; \#f\}$$

At the end, the code generated from this procase consists of two procedures – first one generated from the *?PA.a* action and internally calling the procedures resulted from *#b* and *!SA.c*, and the second one generated from *?PA.d* and calling *#e* and *#f*.

The approach described in the paragraph above works fine except for several cases. In particular, these are: (1) first action of the use case is not a request receive action, (2) a request receive action is in a branch(es), and (3) a request receive action is anywhere inside a loop.

In the case when the first action of the use case is not a request receive action, a special action *INIT* is prepended to the use case and the actions till the first request receive action are enclosed in the procedure call brackets. In the generated code, a procedure generated from the *INIT* action is called automatically before the other actions.

Two other cases cannot be solved directly and require more complex preprocessing. To solve these cases, we have enhanced procases with so called *conditional events*, which allow "cutting" branches of the procase and arrange them in a sequence, but which do not modify the procase syntax.

The conditional events allow to mark branches of the alternatives by a boolean variable (written in the procase just as a name without any prefix symbol followed by a colon, e.g. D:). The variables can be set to *true* via the action written as the variable name prefixed with the $ symbol (e.g. $D) or to false by its name with the $ and ∼ symbols (e.g. ∼$D). At the beginning of each procase, all variables are undeclared.

These events modify the behaviour of the procase in a way that only traces containing the action, which sets the variable to *true*, continue with the branches marked by this variable. When the value of the variable is *false*, the traces continue with the unmarked branches as in unchanged procase.

4.1.1. Branch transformation

First, we show how to rearrange a procase with the request receive action placed in a branch. The general approach of identifying procedures as described above does not work as it would result in nested procedures. To avoid them, it is necessary to rearrange the procase in order to place affected branches sequentially.

We illustrate the branch transformation on the following example:

$$?a; \#b; (\#c; \mathbf{?d}; \#e + \#f; (\#g; \#h + \#i)); \#j$$

The problematic action is *?d* placed in a branch and the whole example is visualized in Fig. 5(a).

The approach of rearranging branches is as follows. Instead of the affected request receive action, the declaration of a conditional event variable is placed. The original action with all subsequent actions till the end of the branch are moved outside the alternative and marked with the chosen variable – depicted in Fig. 5(a ⇒ b).

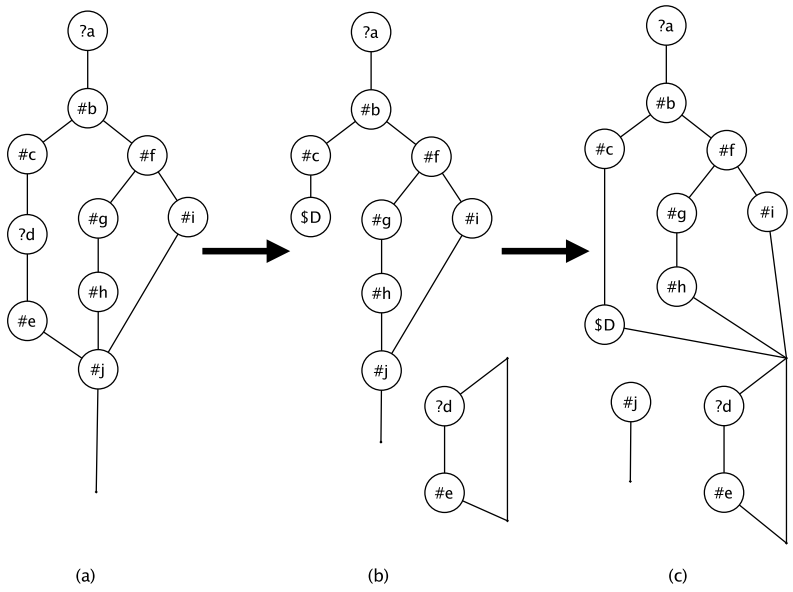


Figure 5. First part of the Branch transformation

The *NULL* action is added as a second branch of the newly created marked branch.

Now, the actions that followed after the original branch action (till the first request receive action) have to be appended to all other branches of this branch action except the branch with the variable declaration – depicted in Fig. 5(b ⇒ c). In addition, these actions are placed at the end of the newly created branch. In a case the variable declaration is placed in more than one branch (i.e. the request receive actions were in more branches), appending of subsequent actions (till the first request receive action) has to be done for all these branches and variables – depicted in Fig. 6(d ⇒ e). This appending guarantees that the resulting procase in Fig. 6(f) generates the same traces as the original one. Now, the general approach of identifying procedures can be applied and yields the following procase:

$$?a\ \{\#b; (\#c; \$D + \#f; (\#g; \#h; \#j + \#i; \#j))\};$$
$$(D : ?d\ \{\#e; \#j\} + NULL)$$

Another example is in Figure 7, which shows the procase of the use case in Fig. 2 that also contains problematic request receive action. Figure 8 depicts the procase after the branch transformation, i.e. it is completely equivalent to the former one and does not contain the problematic branch.

4.1.2. Loop transformation

The transformation of the procases with the request receive action located in a loop action is quite similar to the previous case. Again, the transformation guarantees that the resulting procase generates the same traces as the original one.

The following procase is an example with the request receive action inside the loop:

$$?PA.a; \#b; (\#c; \mathbf{?PA.d}; \#e) * \#f$$

And the resulting transformed procase:

$$?PA.a\ \{\#b; (\#c; \$D + \#f)\};$$
$$(D : ?PA.d\ \{\#e; (\#c + \#f; \sim \$D)\} + NULL)*$$

4.1.3. Unresolved cases

In a case the request receive action is located in two or more nested loops or in a loop nested in branches, the previous two transformations do not work. The procase is then marked as unresolved, excluded from the further processing and has to be managed manually. On the other hand, such use cases are very unreadable (see [8] for suggestions about avoiding extensions of extensions or complex nested loops which results into this problematic procases) and therefore the skipped use cases are candidates for rewriting in a more simple and readable way.

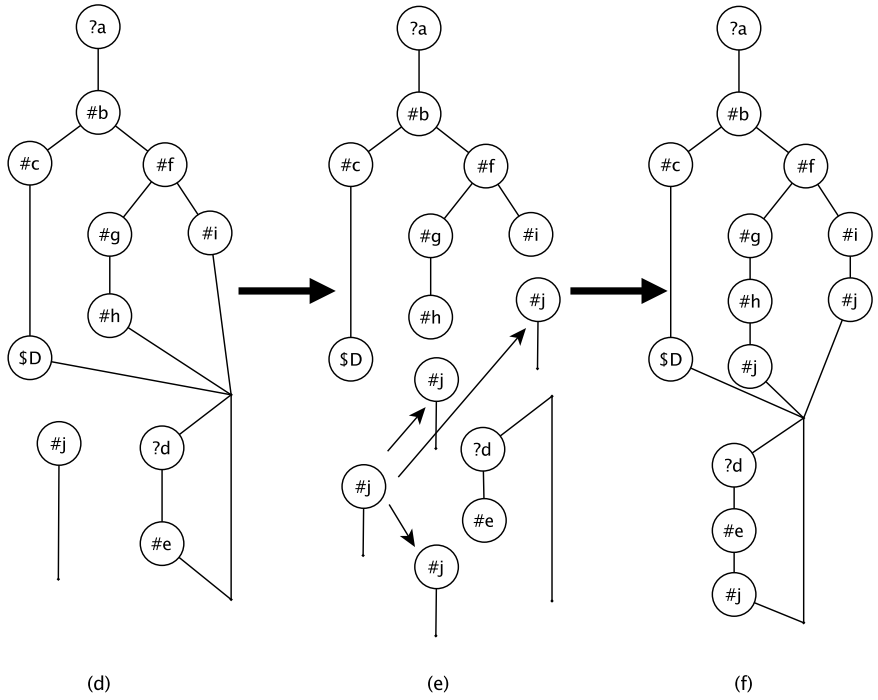


Figure 6. Second part of the Branch transformation

```
?B.submitSelectOffer;
!CS.submitSelectOffer;
!B.reportSystemResponse;
(
  ?B.submitBillingShippingInformationPaymentMethodPaymentDetail;
  !CS.enterBillingShippingInformationPaymentMethodPaymentDetail;
  !B.reportSystemResponse
+
  #validateSystemFail;
  %ABORT
)
```

Figure 7. Procase example before the branch transformation

4.2. Determining Arguments

Once the procases have been preprocessed into sequences of actions grouped as procedure calls, the next step is to determine arguments of the identified procedures, types of these arguments, and how their values are assigned. The arguments are subsequently used as arguments for methods in the final generated code.

In our approach, we are using a fact mentioned in [8] that noun phrases appearing in the use cases are directly related with the domain model elements names. The process of determining arguments is as follows.

First, all noun phrases (which may refer to the data manipulated in the use case step) are extracted by the Procasor from the use case step sentence. In addition, we also take into account *verbs* from the sentence as they can refer to the relations between the conceptual classes in the domain model.

The list of extracted words is then matched against keywords of the domain model (by the keywords we mean names of the classes, attributes, and associations) in order to discover which words are actual attributes and to obtain their types. There are many options to match the keywords – currently in our implementa-

```
?B.submitSelectOffer {
  !CS.submitSelectOffer;
  !B.reportSystemResponse;
  (
    #validateSystemFail;
    %ABORT
  +
    $SBSIPMPD
  )
};
(SBSIPMPD:
  ?B.submitBillingShippingInformationPaymentMethodPaymentDetail {
    !CS.enterBillingShippingInformationPaymentMethodPaymentDetail;
    !B.reportSystemResponse1
  }
+
  NULL
)
```

Figure 8. Procase example from Fig. 7 after the branch transformation

tion we use a simple case-insensitive equality of strings. Such a matching approach can be seen as insufficient but on the other hand, projects commonly follow a chosen terminology (many times explicitly captured in the requirement documents) and therefore our approach is satisfactory in most of the cases.

The determined arguments are compared (by the name and type) with arguments of previous procedures (if they exist) and the already used arguments are copied (their values). If the previous procedures are located in a branch parallel with the *NULL* action they are excluded from processing as they may not be called before the processed one.

Now, the process behaves differently based on a type of the entity, on which the action is called. The types are (i) human user entities (*UE*) such as buyer, seller, etc. and (ii) parts of the system or other computer systems (i.e. system entity – *SE*).

For the trigger action and actions with UE SuD, the unmatched determined types are used as arguments (i.e. parameters which have to be inputted by users). For actions with SE SuD the unmatched determined types are also added as arguments but with default values (during the development of the final application, developers have to provide correct values for them).

4.3. Generating Application

Structure of the generated application employs multiple commonly used design patterns for enterprise applications. Based on these patterns, the generated code is structured into three layers – presentation layer, middle (business) layer, and data layer. In the following text, we refer to objects of the presentation layer as *pages* because the most commonly used presentation layer in contemporary large applications employs web pages, but any type of the user interface can be generated in a similar way.

The middle layer consists of so called *use case objects* which contain the business logic of the application. Also, the middle layer contains *entity objects* where the internal logic (implementation of the basic actions) is generated. The use case objects implement the ordering of the actions and call the entity objects.

We do not describe generation of the data layer, as it is well captured in common UML tools and frameworks (generation of classes from class diagrams etc. – see Sect. 6).

The generation depends on the type of entity – pages are generated for UE while for SE a non-interactive code only. Thus, a page is generated for every action performed by UE (procedure call triggering actions and procedure call internal actions).

If the use case has SuD as UE then elements generated from the actions located inside a procedure call are named with the suffix "X" to allow their easier identification during future development, as in most cases they have to be modified by developers.

Based on combination of the communicating actors (UE vs. SE), the generation distinguishes four cases how the code is generated from a procedure call:

1. If PA and/or SA is UE, then a page is generated for every procedure call triggering action, which is triggered by this UE.
2. If PA and/or SA is SE, an action implementation method is generated in the actor entity object and the action method body contains a call to the corresponding use case object.
3. If SuD is UE, then a method in the corresponding use case object is generated for each procedure call of SuD. The method body calls the actor entity object and redirects to "X" pages, which manage the internal procedure call actions. Internal procedure call actions are generated in a similar way to the request receive action with UE PA – the "X" page and a method inside the "X" use case object are generated. The method inside the "X" use case object is generated as a simple delegation method to the corresponding entity object and redirection to the particular page.
4. And finally if SuD is SE, then inside the corresponding use case object, a method with the body containing the internal procedure call actions is generated.

Figure 9 shows the procase of the Clerk-buys-selected-Offer-on-behalf-of-Buyer use case and Figure 10 overviews all generated elements from the use case.

The following sections describe each type of the generated objects in more details.

4.3.1. Pages

As generated, pages are intended for testing the use cases and are expected to be reimplemented during the further development. A single page is generated for each action interacting with UE. In a case of UE PA, there is a page for every triggering action and, in addition for UE SuD, there is also a page for every procedure call internal action. If the action has arguments which can be inputted then for each of them an input field is generated. Values are assigned by humans during testing of the generated system.

For the UE PA actions, the corresponding pages have a button (an input control element) that allows to continue to the next page, i.e. to continue in the use case (there is only a single button as there is no other choice to continue). On the pages belonging to the UE SuD actions, there are several buttons, which reflect the possibilities of continuation in the original use case. For a sequence of the actions, the page contains the "continue" button; if the next action is a branch action then the page contains a button for each branch (the default button is for the main scenario branch – the buttons for the rest of the branches are labeled by the branch condition label; if the next action is a loop action then the page has a button to enter the loop and another button to skip the loop (following the definition of loop operation).

4.3.2. Use Case Objects

The use case objects contain the business logic (work-flow) of the use case, i.e an order of actions in the main scenario and all possible branches. Bodies of the generated methods differ according to SuD.

UE SuD: As described above, a method in the corresponding use case object is generated for each procedure call of UE SuD. For each trigger action, the method body contains a call to the particular entity object and redirection to a page of the subsequent action. For internal procedure call actions, a similar method body is created in "X" use case object.

SE SuD: A body of the method generated for the procedure call trigger action contains the internal procedure calls. For the SuD internal actions, methods are called on the use case SuD entity object and for request send actions, meth-

```
?CL.submitItemDescription {
    (
        #priceAssessmentAvailable;
        !Sl.providePriceAssessment
    +
        #validateDescription;
        (
            #validationPerformedSystemFails;
            %ABORT
        +
            NULL
        )
    )
};
?CL.enterPriceContactBillingInformation {
    #validateContactInformation;
    !SU.validateSeller
};
?SU.permitSeller {
    !TC.validateOffer;
    (
        #listOffer;
        !Sl.respondUniquelyIdentifiedAuthorizationNumber
    +
        #tradeCommissionRejectsOffer;
        %ABORT
    )
}
```

Figure 9. Clerk-buys-selected-Offer-on-behalf-of-Buyer use case

ods are called on the action triggered entity objects.

The branch actions are generated as a sequence of the condition statements (i.e. *if () ... else if () ...*) with as many elements as branches in the branch action. In each *if* statement, particular actions are generated, while the last *else* statement contains the main scenario actions. A similar construction but with a loop statement (*while*) is created for the loop action.

The number of iteration in the loop statement and choice of the particular branch in the condition statements cannot be determined from the use case. Therefore, the statements are generated with predefined but configurable constants inside the use case object.

4.3.3. Entity Objects

The internal logic of actions is not captured by the use cases neither by the domain model. Therefore, the entity objects are generated with almost empty methods containing only calls to a logger and they have to be finished by developers. For testing purposes, the logging methods seem to be the most suitable ones as designers can immediately check the traces of the generated system.

4.4. Navigation

Navigation (transitions) between the pages is an important part of the application internal logic as it determines the part of the system work flow (the order of procedure calls and sequence of actions). The navigation is derived from the procases as a set of navigation rules. The pages/objects have associated these rules that contain under which circumstances a transition has to be chosen.

In general, the navigation rules are created from the branch actions, loops, and special ac-

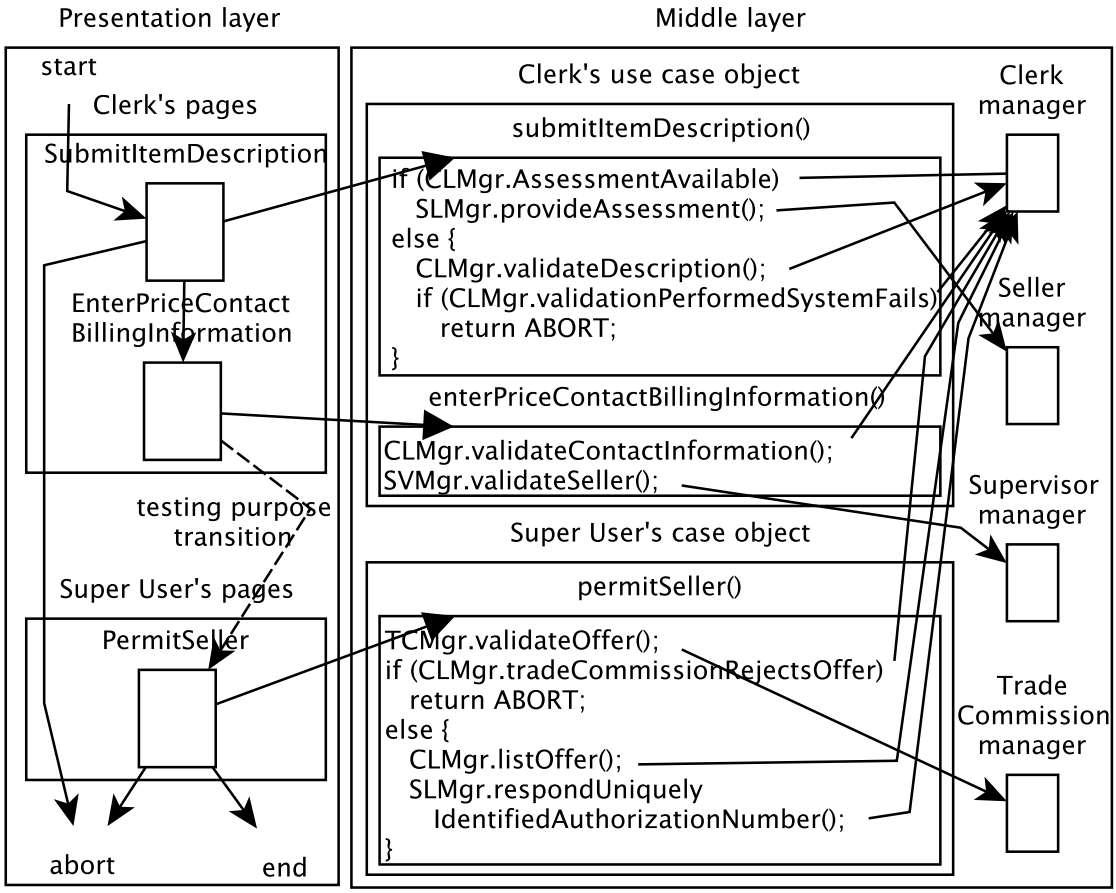


Figure 10. Overview of elements generated from the procase in Fig. 9

tions that can change transitions (aborts, etc.). In a case of the use case with SE SuD, the rules are applied to determine transitions between calls of the actions. In a case of UE SuD, the rules determine how the pages are generated, i.e. which buttons are placed on them.

5. Generated Application Example

To prove that our approach is feasible, we have implemented the proposed generator. As a particular technology for the generated applications, we have chosen the Java EE platform with Enterprise Java Beans (EJB) as the business layer and Java Server Faces (JSF) as the presentation layer. These technologies have been chosen as they are commonly used for large enterprise applications today and thus they can be used as starting point for continuing the application implementation. The generator itself has been written in plain Java.

The generator produces code together with an Ant build file, which can immediately compile and deploy the application to the JBoss application server [7], which allows users to inspect and modify code and iteratively test the application.

Based on the chosen technologies and used design patterns the first two application layers are mapped into the following five tiers. The pages results in two tiers: (i) JSF pages and (ii) backing beans (BB). The middle layer then results into three tiers: (iii) business delegator tier (BD), (iv) Enterprise Java Bean tier (EJB), and finally (v) manager tier (MGR). Generation of the data persistence layer is not currently supported but it is a simple task, which we are plan to add soon (see Sect. 7).

In the rest of this section, we describe in more detail all the generated elements produced from a single use case. As a particular example, the *Clerk submits an offer on behalf of a Seller* use case from the Marketplace example is used. It has the following content.

UseCase: Clerk submits an offer on behalf of a Seller (part)
SuD: Computer System
PA: Clerk

Main success scenario:
1 Clerk submits information describing an item.
2 System validates the description.

Extensions:
2a Validation performed by the system fails.
2a1 Use case aborted.

Sub-variations:
2b Price assessment available.
2b1 System provides the seller with a price assessment.

The Procasor and the preprocessing step of our generator produce the following procase:

```
?CL.submitItemDescription {
    (
        #priceAssessmentAvailable;
        !Sl.providePriceAssessment
    +
        #validateDescription;
        (
            #validationPerformedSystemFails;
            %ABORT
        +
            NULL
        )
    )
}
```

5.1. JSF Pages

All action pages are generated as described in 4.3.1. To allow easy testing, each page displays the use case together with the corresponding procase – both with the highlighted currently processed action and acting entity. The following listing shows a core of the generated JSF page for the use case above.

```
<h:form>
  <h:outputText value="itemDescription : " />
  <h:inputText id="itemDescription"
    value="#{ComputerSystem_
          ClerkSubmitsAnOfferOnBehalfOfASeller_
          ClerkBBean.itemDescription}" />
  </br>
  <h:outputText value="sellerBillingInformation : "  />
  <h:inputText id="sellerBillingInformation"
    value="#{ComputerSystem_
          ClerkSubmitsAnOfferOnBehalfOfASeller_
          ClerkBBean.sellerBillingInformation}" />
  </br>
  <h:commandButton value="submitForm"
    action="#{ComputerSystem_
          ClerkSubmitsAnOfferOnBehalfOfASeller_
          ClerkBBean.submitItemDescription}" />
</h:form>
```

The JSF page has a simple form with an input field for the action argument and a submit button. The triggering action **submitItemDescription** has an argument **itemDescription** which is bound to the use case backing bean variable.

5.2. Backing Beans

According to the JSF framework, the pages are supported by backing beans, which are Java classes containing all variables that can be set by the pages and handling all actions possibly activated by the pages. For the variables, BBs provide setter/getter methods. Also, BBs provide methods for calls to the next tier – business delegators.

The following listing shows the BB's method, which is called when a user submits the form on the page.

```
public String submitItemDescription() {
  try {
    return computerSystem_
      ClerkSubmitsAnOfferOnBehalfOfASellerBD
      .submitItemDescription(getSessionObject()
      . getSellerBillingInformation (), itemDescription,
      getSessionObject());
  } catch (DelegateException e) {
    e.printStackTrace();
  }
  return "abort";
}
```

Before the method call starts, the value of the form's input field is automatically set via the BB's setter method. The value is then used in the method as a parameter of the call to the BD tier.

5.3. Business Delegator

Business delegators are Java classes created on the basis of the Business Delegate pattern [17]. In the generated application, each use case has its own BD, which provides calls to the use case EJBs. Internally, methods of BDs use the Service Locator pattern [17] to locate EJBs.

The method shown in the listing only delegates calls to the use case EJB. The **getBean** method contains code for obtaining a use case bean **LocalHome** interface, creating the bean, and returning the use case bean stub. The stub is then used for the actual call.

```
public String submitItemDescription(String
  sellerBillingInformation,
  String itemDescription, ComputerSystem_
  ClerkSubmitsAnOfferOnBehalfOfASellerSO
  sessionObject) {
  try {
    return getBean().submitItemDescription(
      sellerBillingInformation,
      itemDescription, sessionObject);
  } catch (BeanException e) {
    throw new DelegateException(
      this.getClass().getName() +
      ".submitItemDescription()", e);
  }
}
```

5.4. EJB

The use case objects are generated as stateless session beans. There is no change against the general process described in 4.3.2.

The shown EJB method contains internal logic, i.e. actions located inside the triggered procedure call are executed in this method. The method body structure corresponds to the procase procedure call. Each action is generated as method delegating call to the particular MGR.

```
public String submitItemDescription(String
  sellerBillingInformation,
  String itemDescription, ComputerSystem_
  ClerkSubmitsAnOfferOnBehalfOfASellerSO
  sessionObject) {
  if (Constants.computerSystem_
    ClerkSubmitsAnOfferOnBehalfOfASeller_
      priceAssessmentAvailable) {
    getSellerManager().providePriceAssessment();
  }
  else {
    getComputerSystemManager()
      .validateDescription(itemDescription);
    if (Constants.computerSystem_
      ClerkSubmitsAnOfferOnBehalfOfASeller_
        validationPerformedSystemFails) {
      return NavigationConstants.ABORT;
    }
  }
  return NavigationConstants.CONTINUE;
}
```

5.5. Entity Managers

The entity objects are generated as entity manager classes and they are accessed from the beans, again using the Service locator pattern [17]. The methods of the managers print logs to a console (as explained in Section 4.3.3). We do not show here the generated MGR as its methods contain only these logger calls.

5.6. Additional Elements

In addition to the described elements, there are several additional generated objects that are used across the tiers. Namely, they are *Value objects* and *Session objects*. The former ones are generated and used for each type of arguments of the use case actions, while the later ones hold the value objects among different calls in a use case.

The *INIT* procedure call is generated as the **init** method of the corresponding use case EJB. The special action *%ABORT* is not modeled as an exception but rather as a predefined constant returned from the particular methods.

6. Evaluation and Related Work

To verify our generator, we used it on the Marketplace application described in Sect. 2.1. The generated implementation was compiled and directly deployed to an application server. The implementation consists of approx. 70 classes with 13 EJBs. The complete application has 92 action, from which only 16 actions were generated with wrong arguments and had to be repaired manually (we are working on an en-

hanced method of argument detection – see Sect. 7).

Testing of the generated application discovered a necessity to add one use case, two missing extensions in another use case, and also suggested restructuring other two use cases. All these defects could be detected directly from the use cases but with generated application, they became evident immediately.

Our tool can be also viewed as an ideal application of the Model-driven Architectures (MDA) [11] approach. In this view, the uses cases and domain model serve as a platform independent model, which via several transformations are transformed directly into an executable code, i.e. platform specific model.

Currently, the existing tools usually generate just data structures (source code files, database tables, or XML descriptors) from the UML class diagrams but no interaction between entities (i.e. they handle just the class diagrams) and as far as we know, there is no tool/project that generates the implementation from the description in a natural language. Below, there are several projects or tools that take as an input not only class diagrams but still they work with diagrams and not with a natural language.

The AndroMDA [2] is the generator framework which transforms the UML models into an implementation. It supports transformations into several technologies and it is possible to add new transformations. In general, it works with the class diagrams and based on the class stereotypes, it generates the source code. Moreover, it can be extended to work with other diagram types. A similar generator (made as an Eclipse extension) is openArchitectureWare [12], which is a general model-to-model transformation framework.

In [15], the sequence diagrams together with class diagram are used to generate fragments of code in a language similar to Java. The generation is based on the order of messages captured in the sequence diagram and the structure of the class diagram. There is also a proposed algorithm for checking consistency between these two types of diagrams.

Similarly in [5], Java code fragments are generated from the collaboration and class diagrams. The authors use enhanced collaboration diagrams in order to allow better management of variables in the generated code.

In [16], the use cases are automatically parsed and together with a domain model are used to produce a state transition machine, which reflects behaviour of the system. From the high level view, the used approach is very similar to our solution but they allow for processing only very restricted use cases and thus the approach is quite limited.

7. Conclusion and Future Work

The approach proposed in the paper allows for automated generation of executable code directly from a requirement specification written as use cases in a natural language. Also, we have developed a prototype, which generates JEE applications via the proposed approach.

Applications generated by our tool are immediately ready to be deployed and launched and they are suitable for testing the use cases (i.e. if the requirement specification is complete and well structured) and as a starting point for the development of the real implementation.

The proposed generator has several issues, which suit for further improvements. An important issue is connected with associations among the classes in the domain model. The current implementation correctly handles just one-to-one associations. The one-to-many or many-to-many associations result in the code to lists or arrays and therefore the determination of the arguments is more complex. We plan to solve association limitation by analysis of sentence to determine whether a method argument is the list or object itself. Also, we plan to add a categorization of verbs to allow better management of arguments of the procedures. We plan to employ some platform independent template framework which will enable to generate more configurable system implementation for several platforms. The planned output of the generator then would be a XML file which will be an input

for the employed framework. Finally, we plan to add generation of the data layer to applications.

The required structure of the use cases (based on recommendations in [3]) can be seen as another limitation but we already have an approach, which allows processing of use cases with almost any structure (see [4]) and we are incorporating it to the implementation.

Acknowledgements The authors would like to thank Vladimir Mencl, Jiri Adamek, and Pavel Parizek for valuable comments. This work was partially supported by the Czech Academy of Sciences project 1ET400300504.

References

[1] B. Anda and D. I. Sjober. Investigating the role of use cases in the construction of class diagrams. *Empirical Software Engineering*, Volume 10(3), Jul. 2005.

[2] AndroMDA. http://galaxy.andromda.org.

[3] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Jan. 2000.

[4] J. Drazan and V. Mencl. Improved processing of textual use cases: Deriving behavior specifications. In *Proceedings of SOFSEM 2007, Harrachov, Czech Republic*, Jan. 2007.

[5] G. Engels, R. Huecking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In *Proceedings of UML '99*, Fort Collins, USA, Oct. 1999.

[6] M. Fiedler, J. Francu, V. Mencl, J. Ondrusek, and A. Plsek. Procasor environment: Interactive environment for requirement specification. http://dsrg.mff.cuni.cz/~mencl/procasor-env.

[7] JBoss application server. http://jboss.org.

[8] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, 2nd edition, 2001.

[9] V. Mencl. Deriving behavior specifications from textual use cases. In *Proceedings of WITSE '04, Linz, Austria*, Sep. 2004.

[10] Objecteering software. Objecteering 6. http://www.objecteering.com.

[11] OMG. Model driven architecture (MDA). OMG document ormsc/01-07-01, Jul. 2001.

[12] openArchitectureWare. http://www.openarchitectureware.org.

[13] F. Plasil and V. Mencl. Getting "whole picture" behavior in a use case model. In *Proceedings of IDPT*, Austin, Texas, USA, Dec. 2003.

[14] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, Volume 28(11), Nov. 2002.

[15] L. Quan, L. Zhiming, L. Xiaoshan, and H. Jifeng. Consistent code generation from UML models. UNU-IIST Rep. No. 319, The United Nations University, Apr. 2005.

[16] S. S. Somé. Supporting use cases based requirements engineering. *Information and Software Technology*, 48(11):43–58, 2006.

[17] Sun Microsystems. Core J2EE patterns: Best practices and design strategies. http://java.sun.com/blueprints/corej2eepatterns.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Tool Based Support of the Pattern Instance Creation

Ľubomír Majtás*

*Faculty of Informatics and Information Technologies, Institute of Informatics and Software Engineering, Slovak University of Technology

majtas@fiit.stuba.sk

Abstract

Patterns introduce very useful way of improving the quality of the software development process. Nowadays modeling tools and techniques provide some kind of support for modeling with pattern instances, but these are often based on manual pattern creation and connection to the rest of the model. Our approach presents the support of pattern instance creation on the model level in semi automatic way that simplifies the whole process. The main idea of this approach is that the developer should assign the domain dependent parts of pattern and specify the requirements over the pattern variants. The rest of the pattern instance is to be created by the machine.

1. Introduction

Pattern introduction [1] had large asset for more areas. Probably the most familiar pattern's fulfillment in the software engineering was introduced by the work of GoF [12], where the authors identified and in detail described 23 design patterns. Their description of each pattern contains the verbal description of its main idea, the example of its appropriate usage (including the source codes), the description of solution it offers and discussion about its alternatives and consequences of its usage. The main part of description is presentation of the pattern model according to the OMT/UML diagrams. The authors provided patterns' explanations by examples and textual description so the catalog means a useful knowledge base for software professionals. On the other hand they did not try to present any "computer friendly" knowledge that could be basis for automation of typical pattern processes which are:

- Creating of pattern custom instances,
- Validating existing pattern instances,
- Identification of pattern instances in existing codes.

To solve this drawback, there were presented many other works that extend the original catalog by the different models that are trying to capture the core structure of patterns, e.g. [11], [13], [7].

In this paper we would like to introduce the approach that would support developers in their work with pattern instances. Nowadays modeling tools and techniques provide some kind of support for modeling with pattern instances, but these are often based on manual pattern instance creation and connection to the rest of the model. Our approach presents the support of pattern instance creation at the model level in semi automatic way that simplifies the whole process. The core idea of the approach is that the developer should assign the domain dependent parts of pattern and specify the requirements over the pattern variants. The rest of the pattern instance should be generated automatically. In this paper we will analyze the processes taking place while creating the pattern instance. We will identify the places where can be this process

automated. Finally we will provide our approach of automation in a way that will support but not limit the developers.

2. Process of the Pattern Instance Creation

Process of the pattern instance creation means the application of the solution offered by the pattern to the environment of the developed software system. The inputs of this process are the actual environment of the developed software and the general description of the pattern. As the output we consider modified software system extended by correctly created instance of the pattern.

We distinguish two activities that are necessary to follow out while creating the pattern instance [15]: abstract and general pattern instance needs to be concretized and specialized. At the first moment both activities seem to be similar, but it is not so. Each one moves the first idea of the pattern application to the final instance, while it is necessary to follow up both, to be able to declare the pattern instance as the correct one. Differences between these activities are presented in the Figure 1, where the degrees of generality and abstraction are being presented in two dimensional space (degree of generality horizontally, degree of abstraction vertically).

The created instance is becoming more concrete when it contains more building blocks creating the correct instance. To the beginning abstract idea of the pattern application there are subsequently being added classes, their attributes, methods and relations until instance becomes complete. Specialization of the instance means the movement of the general pattern description to the context of the developed system. The specialization follows such modifications of the pattern instance that make the instance domain specific and subsequently specific for the current software system. As the examples of the specialization steps we can consider definitions of roles' participants count, naming of the participants or creating the relations between participants according to the domain.

In our approach we look for possibilities for automation of the pattern instance creation. We see higher potential in the process of concretization than in process of specialization. The specialization is based on the ability of developer to move the pattern to the particular target software environment. It can be seen as a domain based pattern description, what we consider as almost impossible to be performed by the machine. There can be found only minimal space for automation of this process. On the other hand, there is a potential for tool based support of concretization process. When the pattern instance is correctly specialized, its concretization is often based more on pattern structure description than on developer's skills. It means that there is a space for automation of this process.

2.1. Pattern Roles, Domain Dependency

Patterns are often being described as a collection of cooperating roles. These roles can be often divided into two groups: roles dealing with the domain of the created software system (domain roles) and roles performing the pattern's infrastructure (infrastructure roles). The domain roles can be considered as the "hot spots" while they can be modified, added or deleted according to the requirements of the particular software environment. The roles performing the pattern infrastructure are not changing frequently between the pattern instances. Their purpose is to glue the domain roles together to be able to perform desired common functionality.

One of the main contributions of the whole pattern approach is that it allows thinking at the higher level of abstraction. Developers do not have to always keep in mind all details about the solution, they can work with the pattern instance as with single unit hiding unnecessary complexity. When the developer thinks about applying the pattern to the project, first thing he needs to decide is how to connect pattern instance to the context of the software. He does so be specifying the domain roles' participants. The other issues are often second-rate at that moment. Table 1 describes selected patterns and

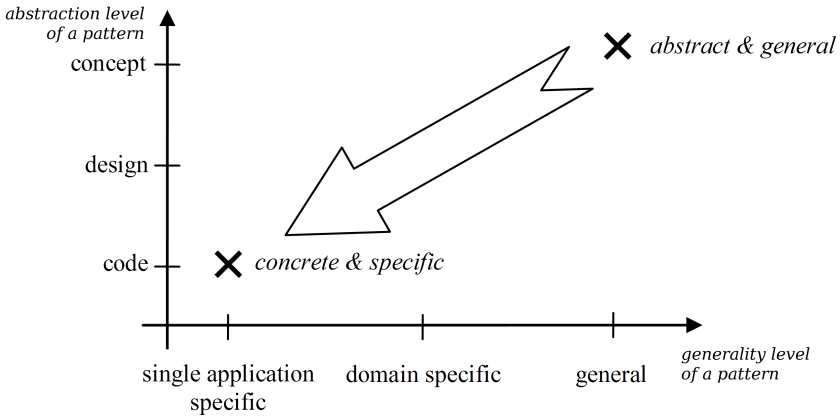


Figure 1. Two dimensional space of generality and abstraction [15]

Table 1. Specialization of the domain dependent pattern roles

Pattern	Domain dependent roles	Description
Composite	Leaf and its Operations	Leafs and their operations provide all domain dependent functionality. Everything else is just infrastructure allowing the hierarchical access to the leaf instances.
Chain of Responsibility	HandleRequest, Ancestor	The domain dependent is the business logic processing the event and the ancestor to which should be the unprocessable event passed.
Decorator	Concrete Component, Concrete Decorator	The domain dependent are the Concrete Component (which often exists before Decorator pattern application) and functionality of Concrete Decorator participants that provide extended functionality to the Concrete Component.
Flyweight	Concrete Flyweight	Concrete Flyweight provides all domain dependent functionality. The rest is infrastructure for storing instances in memory and providing access to them.
Proxy	Real Subject, Proxy	The domain dependent is the Real Subject (which often exists before Proxy pattern application) and functionality of Proxy participants that provide access to the Real Subject.

their roles from the perspective of the domain dependency.

2.2. Pattern Variability

As the pattern variability we understand the possibility to provide patterns functionality in slightly different ways. Each variant of the pattern has its own pros and cons and therefore the decision which variant to select does not need to be easy. Selecting the proper variant is part of pattern instantiation process, when we are customizing instance according our needs. By the variability we do not understand definition of participants playing defined roles.

General understanding of the patterns does not always satisfy the ideas of their authors. Many developers understand design patterns only as constant templates with strictly specified purpose of each class, attribute or method. However the original idea was to discuss a problem and offer a solution. Examples provided with pattern description were never meant as the only best solutions. Their purpose was to

Table 2. Examples of possible variants of pattern instances

Pattern	Variability description
Abstract Factory	Class structures do not differ much. Majority of variants are dealing with realizations of ConreteFactories: they can be implemented normally or as Singletons, they can employ Factory Method or Prototype patterns.
Adapter	There are more different variations how can Target, Adapter and Adaptee communicate together.
Bridge	Possible variants: Omitting the Implementer interface in case of the only one ConcreteImplementor.
Chain of responsibility	References to the successor can be maintained commonly in Handler or custom for each ConcreteHandler. Handler can be a class with default forwarding functionality or just an interface.
Flyweight	The Flyweight interface can be omitted.

provide hint, to show one of many possible ways of idea realization.

When we look closer on the patterns from the GoF catalog, we can distinguish differences of examples generality between patterns. On the one side stands Singleton. It is very simple with accurate example that is not keeping much space for different variations while it prescribes all desired functionality. On the other side we can see patterns such as Memento. Their examples are very general; they do not provide expected functionality but only briefly draft the solutions. Their concrete instances can be far different from the presented examples. In the middle of these extremes stands majority of the patterns. Their examples are able to provide desired functionality while they are keeping a space for their customization. Typical representatives are Composite, Observer or Decorator.

From the perspective of tool based support of instantiation, it is very difficult to create support for instantiation of pattern with very general examples. It would be very difficult (if it is even possible) to automatically instantiate fully functional Memento that would fit to the rest of the system. On the other hand there are minimal difficulties for pattern with strictly defined structure keeping minimal space for variability. Pattern such as Singleton can be automatically instantiated with minimal efforts. The simple template based instantiation would be sufficient. For the majority of patterns the simple template based approach cannot cover all known variability, such approach cannot be considered as sufficient. In this case we need to employ approach that is based on templates that are created according the user needs. In Table 2 we present examples of possible variabilities of selected GoF design patterns.

2.3. Pattern Instantiation Support in CASE Tools

Many existing CASE tools are trying to provide some kind of support in pattern instantiation process. The level of such support differs. Often they allow inserting of example pattern instances to the model. The others can run wizards through which developer can specify the participant count of the selected roles and specify the name for each one. In general the support is based on single template, where the developer can or cannot specify the participants before the creation of the instance. Advanced CASE tools are tacking the information about pattern occurrence (often by UML Collaboration element). This helps the further developers to identify the pattern with minimal effort and thereby it reduces the risk of instance damage that can happen by improper modifications in later project phases (e.g. maintenance). However, these tools do not try to automate the process pattern instantiation – participants of all roles need to be specified by developers. Alike the support for other kinds of customizations is often omitted; it is left to developers' knowledge and experience to modify the instance according their needs.

Employing the automation of pattern instantiation in CASE tools can lead to the following benefits:

- Developers do not need to perform typical modifications manually. By minimizing the effort that needs the developer to perform to create pattern instance or by giving him possibility to select the proper pattern variant they can save time and avoid mistakes, so the instantiation process becomes more effective.
- Developers do not need to know all pattern complexity or inner structure. They can focus on the domain dependent context of the pattern; they "do not need to care" about the rest. This can help the inexperienced developers with pattern application, and in this way support them to utilize patterns in their everyday work.
- Developers can be informed about possible variants. Sometimes developers do not have to know about existence of different pattern variant. When they are informed immediately about more possibilities they can choose most proper variant without former knowledge about it. Developers are able to get best from the pattern application.

3. Our Approach of the Tool Based Support

In the previous sections we have described why should we consider the tool based support for the pattern instance creation and where are the spots for the automation. In this section we will describe how can be such automation provided. We focus on two different ways of support. The first one is dealing with the process of instance creation, it lets the developer to define domain based participants and automatically supplement infrastructure participants to form a valid instance. The second one provides support for pattern variability; it informs the developer about possible variants of the pattern, asks for desired ones and automatically reasons the valid configuration according the developer's choice. The result of this step is the role based model of the proper pattern configuration, which comes as an input for the first mentioned support.

3.1. Pattern Inner Structure Description

To be able to provide machine based pattern processing, we require precise models of the patterns' inner structures. We are using custom role based models which are defining the collaborations between the roles that perform the predefined functionality. As the roles we do not consider only ones typically played by classes but also ones which participants are attributes or methods. Only the definitions of the roles do not capture whole pattern inner structure and therefore cannot be used as the blueprint for the machine based pattern instance creation. The very important parts of the pattern structure are the definitions of the inner structure constraints. These constraints associate roles that are somehow linked together. Example of such constraints can be clearly seen in the Abstract Factory pattern where the roles createProduct() of Abstract Factory and Abstract Product are linked together. It means that there has to be the same count of participants of this role where each participant of the createProduct() role is responsible for creation of the appropriate participant of the Abstract Product. The exact definitions of constraints are very important in the process of machine based pattern instantiation while they help to specify the count of all participants and set the proper links between them. We have identified and capture in our models the following relationships which are bases of constraints:

1. Inheritance – inheritance between classes,
2. Association – associations between classes,
3. Overriding – in case of inheritance where the one method role overrides the other method role,
4. Method delegation – one role invokes other role to delegate the functionality,
5. Instance creation – role creates the instances of other role,
6. Class linked with its members – role which participants are regularly classes and can be played by more than one participant needs to be explicitly linked with its method and attribute roles.

Some roles are part of definition of more than one constraint. For example in the Abstract Factory pattern the count of participant of role Concrete Product is dependent on count of participants of roles Concrete Factory and Abstract Product. We say that the dimension [13] of the role Concrete Factory is two because it is part of two different constraints.

3.2. Algorithm of the Pattern Instance Creation

Our process of pattern instance creation is based on supplementing the incomplete pattern instance defined by developer. As the inputs the algorithm requires proper role based pattern model (according to the previous section) and the partially created pattern instance containing some participants of the domain role. The algorithm progressively adds the missing participant to comply the pattern's inner structure description and all constraints of the pattern. The output of the algorithm is the model of pattern instance containing all participants that are meeting all constraints.

The algorithm stores information about all participants which are part of the pattern instance at the moment. Moreover it stores information about instances of all constraints connecting the linked corresponding participants. Some roles are present in more constraints (their dimension is more than one) what causes that instances of constraints overlap over the participants of such roles. Therefore the algorithm stores the information about instances of constraints in the n-dimensional structure where n is the maximum dimension of all pattern roles (the GoF pattern do not have roles with dimension more than 2). We call this structure Participant Constraint Matrix (PCM). Each dimension of this matrix corresponds to one pattern constraint. Lines in this dimension represent the instances of constraints. These instances of constraints link corresponding participants according to the constraint. Lines cross in the places of more dimensional participants. Examples of partially filled Participant Constraint Matrix are depicted in the Figures 2, 3 and 4.

In the following section we describe the steps of the algorithm creating the pattern instance. We will describe the algorithm also on example, each step will contain example of execution results of this step. In our example we will create the instance of the pattern Composite. As the input we get the incomplete instance containing only partial definitions of two participants of the domain role Leaf: Leaf1 containing the Operation1() and Leaf2 containing the Operation2(). The algorithm will create the correct instance according these inputs. The algorithm takes the following steps:

1. Add participants of non constrained roles. Create the participants of the roles that are not concerned in any constraint. For each role create exactly one participant and name it as the role. Create the links that are related to the new participants.
   Example: Add the participants of roles Component, Composite, Composite's childs and links between them: generalization and association.
2. *Create the empty Participant Constraint Matrix.* Create the empty PCM according to the definition of pattern constraints.
3. *Initialize the PCM according to the current of the pattern instance.* Fill the PCM with already created the participants.
   Example: Fill the PCM as depicted in the Figure 2.
4. *while (the PCM contains empty fields)*
   a) *Add participant to fill one empty field of PCM.* Select one empty field of the PCM and create the participant that will fit to this field. When selecting the empty fields, start with class participants and continue with association and method participants. Prefer empty fields with lower dimension.
      Example: In the first iteration create the participant of role Component's Operation().
   b) *Add information related to the added participant.* Fill the information about the new participant and add connections

			Leaf members constraint	
			<<Leaf>> Leaf1	**<<Leaf>>** Leaf2
Operation() overriding constraint	**<<Component's operation>>**	**<<Composite's operation>>**	**<<Leaf's operation>>** Leaf1's Operation1()	**<<Leaf's operation>>**
	<<Component's operation>>	**<<Composite's operation>>**	**<<Leaf's operation>>**	**<<Leaf's operation>>** Leaf2's Operation2()

Figure 2. Initial Pattern Constraint Matrix for incomplete instance of pattern Composite

			Leaf members constraint	
			<<Leaf>> Leaf1	**<<Leaf>>** Leaf2
Operation() overriding constraint	**<<Component's operation>>** Component's Operation1()	**<<Composite's operation>>**	**<<Leaf's operation>>** Leaf1's Operation1()	**<<Leaf's operation>>**
	<<Component's operation>>	**<<Composite's operation>>**	**<<Leaf's operation>>**	**<<Leaf's operation>>** Leaf2's Operation2()

Figure 3. Extended of Pattern Constraint Matrix after adding Component's Operation()

			Leaf members constraint	
			<<Leaf>> Leaf1	**<<Leaf>>** Leaf2
Operation() overriding constraint	**<<Component's operation>>** Component's Operation1()	**<<Composite's operation>>** Composite's Operation1()	**<<Leaf's operation>>** Leaf1's Operation1()	**<<Leaf's operation>>** Leaf2's Operation1()
	<<Component's operation>> Components Operation2()	**<<Composite's operation>>** Composite's Operation2()	**<<Leaf's operation>>** Leaf1's Operation2()	**<<Leaf's operation>>** Leaf2's Operation2()

Figure 4. Pattern Constraint Matrix for complete instance of pattern Composite

with the other participants that are related to the new one, e.g. generalization, overriding, association, delegation, etc. Example: Connect the participant with Leaf's Operation1() with overriding relationship. Name the participant Operation1() because the overriding relationship needs the same name of the linked participants. Extended PCM by this step is depicted in the Figure 3.

After successful execution of the algorithm the pattern model of the instance is created. It complies with all rules and constraints coming from the pattern description. The created model keeps information about the role that participants play and therefore can be used for further source code generation of the pattern instance. PCM for the complete pattern instance is depicted in the Figure 4.

3.3. Pattern Variability

As mentioned in previous sections patterns are not simple units with the only one valid template. Most of them are highly customizable allowing changes in their example templates in many different ways. In this section we describe the approach of employing the variability to the instantiation process. The result of this step is role based model describing the customized pattern template that stands as an input for previous algorithm.

3.3.1. Variability Modeling

To capture possible variability, we are employing feature modeling technique that was originally designed for the product-line engineering [6]. It is important for capturing and managing commonalities and variabilities in product lines throughout all stages of product-line engineering. In early stages it is used for scoping of product line (i.e. deciding which features should be supported by a product line). In product-line design, the variation captured from feature models are mapped to product-line architecture common for all parallel product lines. In the product development, feature models can drive requirements elicitation and analysis, help in estimating development cost and effort, and provide a basis for automated product configuration. Feature models are also important in generative software development which is trying to automate application engineering based on system families.

In our approach we use the feature models to capture possible variants of the patterns. The model depicted in the Figure 5 presents selected variabilities of the Composite pattern. It says for example that participant of the role Component can be either the interface or the abstract class or that the processing method of the Composite's children role can be omitted, present only in the Composite or in the whole structure. The feature model also presents relations between the features. For example it is not possible to omit these processing methods when the Composite's children is private attribute. Such configuration would disable the whole pattern's functionality because the structure would become unmodifiable.

The presented model is considered only as an example. It does not cover all the variabilities such as the way of Composite's children collection realization.

3.3.2. Configuration Reasoning

When creating a pattern instance, developer needs to specify his requirements dealing the variability. The target is to specify whole feature selection (also called product configuration), which is a group of desired functional capabilities that constitute a complete configuration of an application and adhere to the constraints specified in the feature model. We do not want to force the user to provide information about each feature whether he wishes to employ it or not. We give him a chance to specify which features he wishes to employ and the rest of the configuration is set by the tool. The final configuration has to fulfill all constraints defined by the feature model, so if the developer's requirements do not meet these constraints or do not allow creation of valid configuration, the developer has to be asked to change his preferences.

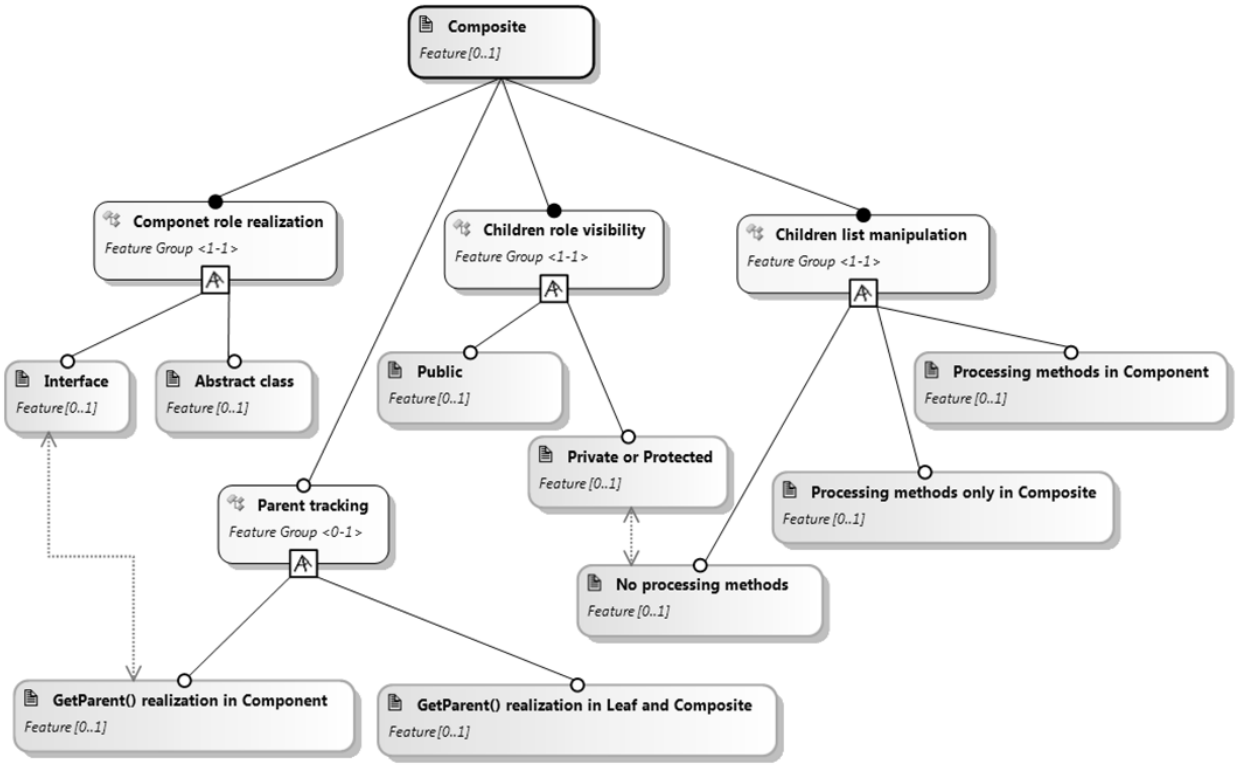


Figure 5. Feature model example capturing the Composite pattern variability

To reason a valid configuration we transform the feature model and partial configuration based on developers preferences to Constraint Specification Problem (CSP) environment and apply existing CSP solver to reason final configuration or to notify us about impossibility to finish this task. To create the CSP from a feature model we apply transformation rules specified in [2]. As the CSP solver we chose Choco CSP [4] which is an open source Java based software.

3.3.3. Final Model

When the configuration is set up, we are able to provide concrete role based model of the pattern instance. The variants represented in the feature model have several possible impacts to the final instance, while they can:

- Prescribe the role based model
  - Specify the occurrence of the role, whether participants of the role should be part of pattern instance or not.
  - Specify the position of the role. For example specify, whether the operation should be present in parent class or only in child classes.
- Define the implementation aspects relevant for code generation
  - Specify the form of participants' realizations. For example they can specify whether class role would be played by class or interface or whether list will be realized as array, linked list, map or something else.
  - Specify the participants' visibilities: Public/Private/Protected.

The definition of the roles occurrence or positions prescribes the output role based model. It is provided by reduction of the general pattern template containing all variability roles. This template contains all roles including the conditions when should be these role applied (which feature needs to be part of the configuration to activate the variability role). The Figure 6 contains such template for the Composite pattern according the features presented in the Figure 5. The variability roles are filled grey and the activating features are placed next to them as a bold text. According the current configuration, the variability roles with features that are not contained in the configura-

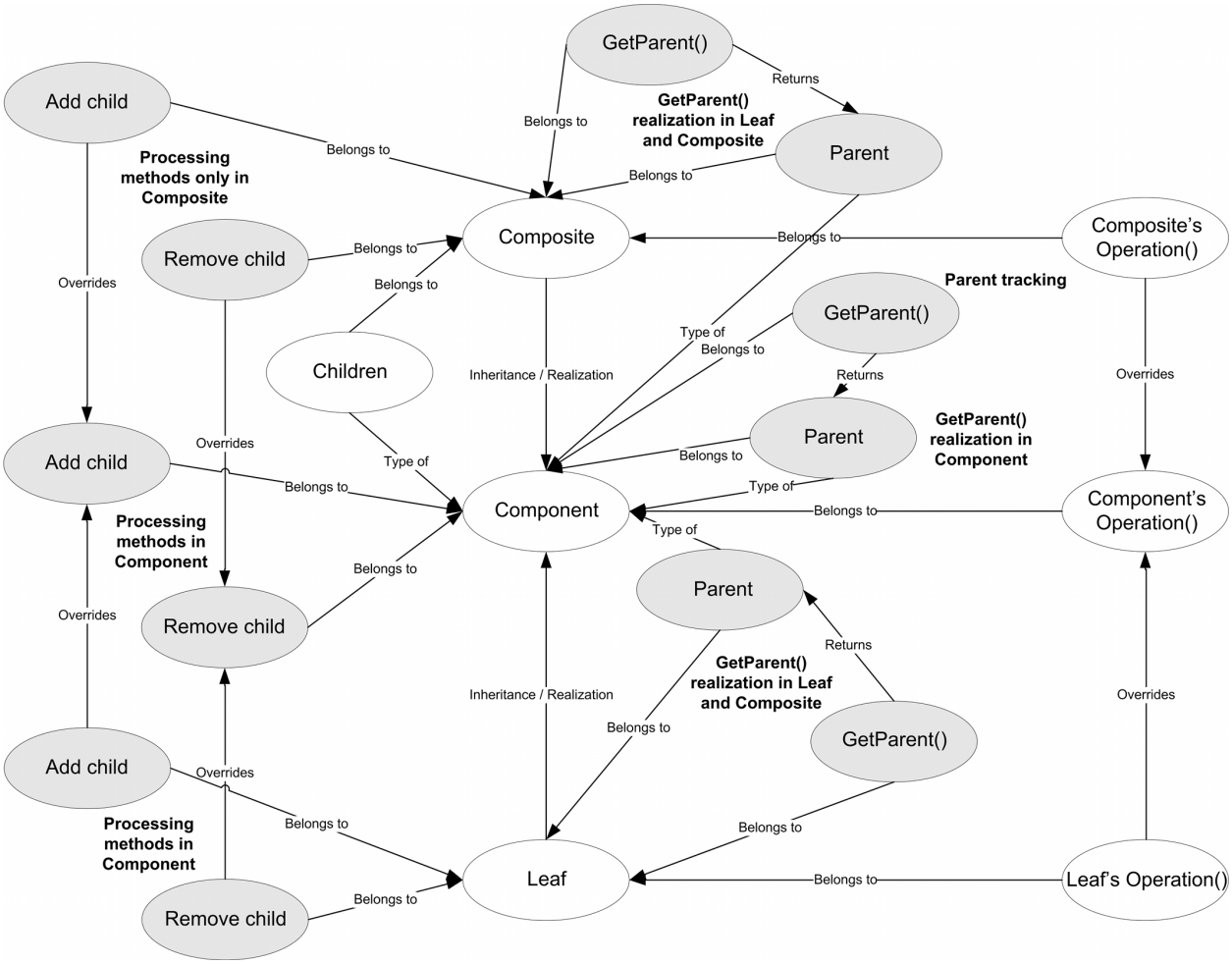


Figure 6. Role based template of the Composite pattern extended by variability roles

tion will be omitted, the rest will form the final role based model according the input configuration.

As an example, when we take the Composite's feature model from the Figure 5 and select the following variants: Parent tracking – GetParent() realization in Component and the Children list manipulation – No processing methods. The final role base model representing the configuration received from the previous inputs is depicted in the Figure 7.

3.4. Overall Instantiation Process

Presented approaches form a complex process of computer aided pattern instantiation deliberating the pattern variability. It consists of the following steps:

1. Inquire the user about pattern he wishes to instantiate;
2. Inquire the desired domain participants;
3. Inquire the desired variants of the pattern;
4. Reason pattern configuration;
5. Create the role based model for the reasoned configuration;
6. Supplement all missing participants from the incomplete user specification according the actual role based model;
7. Create the instance of the pattern.

In general the user is inquired for the domain dependent information and customizations while the tool creates the pattern instance satisfying the user needs. The architecture scheme of the overall approach is sketched in the Figure 8. The general input for the entire process is role based pattern model containing all variability roles. Variabil-

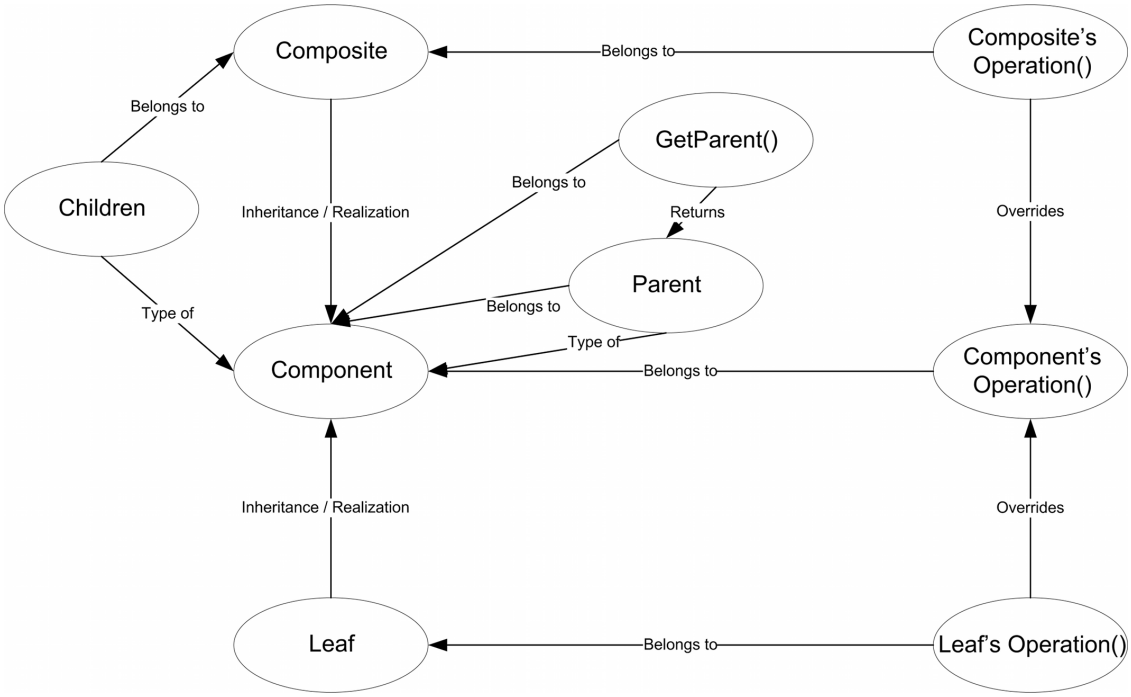


Figure 7. Example of output role based model according the custom configuration

ity support module reduces this model into the concrete role based model for the custom pattern configuration inferred from developer's variants selection. Pattern instance creation module takes this model and according to it and developer's specification of domain dependent participants creates the final pattern instance. This final instance reflects the developer's variants selection and pattern domain specialization.

3.5. Realization

The presented approach was partially implemented and verified. It was realized as the plug-in of the Rational Software Modeler which is based on open source platform Eclipse. The Figure 9 contains screenshots of the model before and after the execution of the overall pattern instantiation process. As the inputs for this scenario we have used the inputs of the examples presented in former sections.

4. Related Work

Different approaches of automating the pattern utilization in software projects were introduced by the other authors. Ó Cinnéide et. al. [5] have presented a methodology for the creation of behavior-preserving design pattern transformations and applied this methodology to GoF design patterns. The methodology is taking place in refactoring process when it provides descriptions of transformations to modify the spots for pattern instance placement (so called precursors) by the application of so called micropatterns to the final pattern instances. While Ó Cinnéide's approach is supposed to guide the developers to pattern employment in the phase of refactoring (based on source code analysis), Briand et. al. [3] are trying identify the spots for pattern instance in design phase (based on UML model analysis). They provide semi-automatic suggestion mechanism based on decision tree combining evaluation the automatic detection rules with user queries.

There exist several approaches introducing their own tool based support for the pattern instantiation. El Boussaidi et. al. [10] present model transformations based on Eclipse EMF and JRule framework. Wang et. al. [16] provide similar functionality by XSLT based transformations of the models stored in XMI-Light format. Both approaches can be considered as

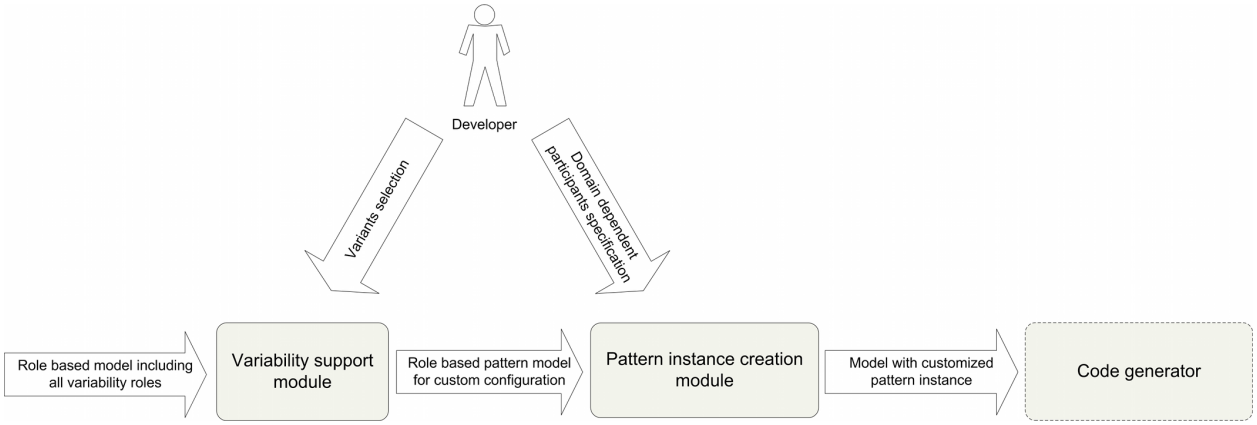


Figure 8. Architecture overview of the overall approach

the single template driven while they are focusing most on the transformation process and do not set a space for the pattern customizations. More advanced method was introduced by Mapelsden et. al [14]. Their approach supports instance configuration by specifying the role participants including those playing more dimensional roles. All of these approaches are based on strict forward participants generation – participants of all roles are created according the single template. Our approach accentuates on collaboration between the developer and the CASE tool. We do not intent to create all pattern participants. We let the developer define the ones he needs and subsequently we infer and create the rest ones to form a valid instance. We do not force the developer to our solution, we let the template and the final instance be customized according the developers' needs.

All the former approaches were focusing on the creation of pattern instances. The ones presented by Dong et. al. presume the presence of the pattern instances in the model. They are providing the support for the evolution of the existing pattern instances resulting from the application changes. The first one [8] implementation employs QVT based model transformations, the other one [9] does the some by the XSLT transformations over the model stored as XMI. However, both are working with the single configuration pattern template allowing only the changes in presence of hot spots participants. Other possible variabilities are omitted.

We have not found any approach regarding the feature modeling application in pattern instantiation area. The feature models were successfully employed in other areas, for example in automation attempt of enterprise application configuration presented by White et. al. [17].

5. Conclusion and the Future Work

In this paper we have presented our approach dealing with a tool based support for pattern instance creation. Our key concept was to create such methodology that would help the developers with application of the pattern solution to their software but allow them to customize the pattern according their needs. We were trying to handle two different courses while more generative parts often mean less space for customization and vice versa. We believe that our approach balances these opposing courses into final solution in a way that forms useful a tool for developers interested in pattern employment.

In the future we would like to extent the created pattern instance model with behavioral information. The correctly created instance would be represented class diagram together with sequence diagram. The main building blocks of such behavioral model will be method invocation and delegation, instance creation together with structural blocks such as condition or iteration over collection. Also we would like to prepare definitions of more GoF design pattern to evaluate the algorithm on the larger scale.

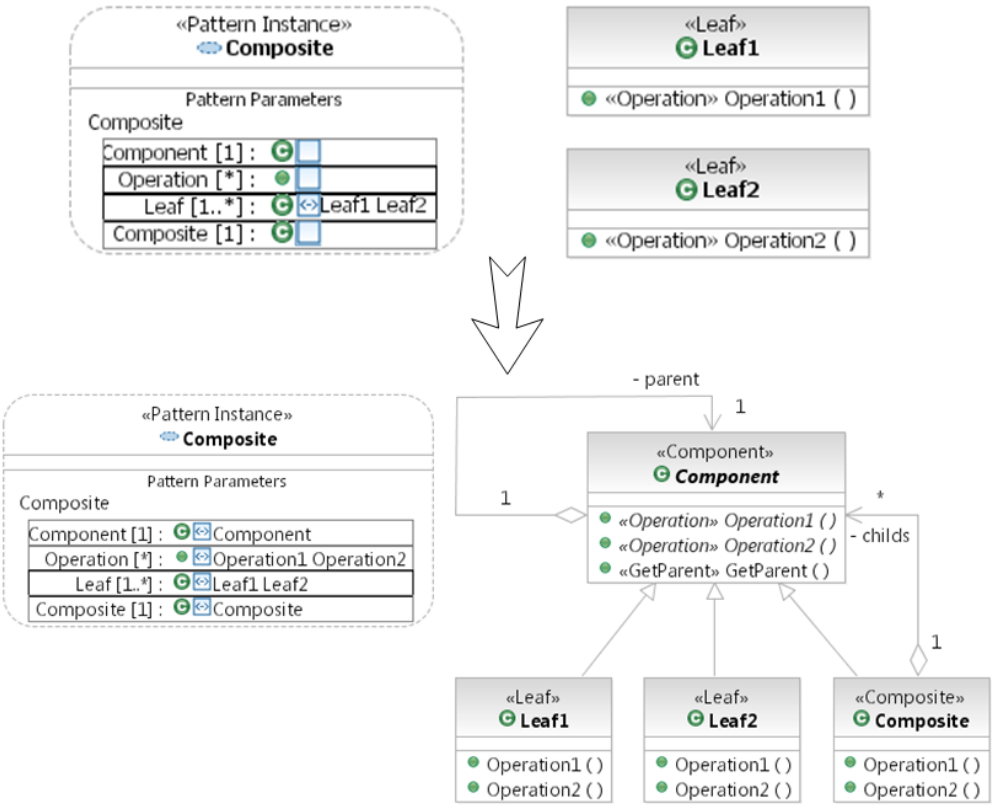


Figure 9. Screenshots of models before and after the overall process execution

We are thinking about other patterns that can be input for the algorithm. It could be applicable on all patterns that are at the design level of abstraction and their inner structure can be described by relation based constraints. Candidates for such patterns are for example the J2EE design patterns.

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, August 1977.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*. Springer, 2005.

[3] L. C. Briand, Y. Labiche, and A. Sauve. Guiding the application of design patterns based on UML models. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 234–243, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Choco constraint programming system. http://choco.sourceforge.net/.

[5] M. Ó Cinnéide and P. Nixon. Automated software evolution towards design patterns. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 162–165, Vienna, Austria, 2001. ACM.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[7] J. Dietrich and C. Elgar. A formal description of design patterns using owl. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 243–250, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. Dong and S. Yang. Qvt based model transformation for design pattern evolutions. In *IMSA '06 : Proceedings of the 10th IASTED international conference on Internet and multimedia systems and applications*, pages 16–22, 2006.

[9] J. Dong, S. Yang, and K. Zhang. A model transformation approach for design pattern evolutions. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 80–92, Washington, DC, USA, 2006. IEEE Computer Society.

[10] G. El Boussaidi and H. Mili. A model-driven framework for representing and applying design patterns. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 97–100, Washington, DC, USA, 2007. IEEE Computer Society.

[11] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30(3):193–206, 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[13] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. Mapelsden, J. Hosking, and J. Grundy. Design pattern modelling and instantiation using dpml. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[15] M. Smolárová, P. Návrat, and M. Bieliková. A technique for modelling design patterns. In *JCKBSE '98: Proceedings of the Knowledge-Based Software Engineering*, pages 89–97. IOS Press, 1998.

[16] X.-B. Wang, Q.-Y. Wu, H.-M. Wang, and D.-X. Shi. Research and implementation of design pattern-oriented model transformation. In *ICCGI '07: Proceedings of the International Multi-Conference on Computing in the Global Information Technology*, page 24, Washington, DC, USA, 2007. IEEE Computer Society.

[17] J. White, D. C. Schmidt, K. Czarnecki, C. Wienands, G. Lenz, E. Wuchner, and L. Fiege. Automated model-based configuration of enterprise java applications. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 301, Washington, DC, USA, 2007. IEEE Computer Society.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Transformational Design of Business Processes in BPEL Language

Andrzej Ratkowski*, Andrzej Zalewski*, Bartłomiej Piech**
*Instituite of Control and Computation Engineering, Warsaw University of Technology
**Department of Electronics and Information Technology, Warsaw University of Technology
a.ratkowski@elka.pw.edu.pl, a.zalewski@ia.pw.edu.pl, b.piech@elka.pw.edu.pl

Abstract

A transformational approach to the design of executable processes in Business Process Execution Language (BPEL) is presented. It has been built upon the transformations of business processes accompanied by a formal approach based on process algebras used to verify the behavioral equivalence of business processes. The initial business process can be denoted in BPEL, then a series of transformations is executed upon it. The process resulting from the transformation is verified whether it preserves behaviour denoted by the process being transformed. The transformations improve non-functional properties of the process (performance, modifiability, granularity, maintainability) but do not change its original behaviour. The transformations are steered by Architecture Trade-off Analysis Method (ATAM) that shows the direction of changes and helps an architect to decide which of them to apply. An example of the application of our approach in real-life business process design has also been presented. The paper presents general idea of the design process, theoretical basis of the method as well as experimental verification of the approach and a tool implemented to support the method.

1. Introduction

The following paper presents the concept of design method that is a subject of PhD thesis written by Andrzej Ratkowski under Prof. Krzysztof Sacha's supervision. The article is an extension of a previous paper [26].

The ability to define and execute business processes seems to be one of the most important advances introduced by the research and commercial developments on Service-Oriented Architectures (SOA). The worlds of business modelling and software systems development have never been closer to each other – it is now possible to express software requirements in terms of services and business processes composed of them. BPEL have become a standard for defining executable business processes. This in turn triggered an extensive research on the modeling and verification techniques suitable for those processes.

The approaches presented above, as well as the verification techniques, can indicate absence or existence of certain flows in BPEL processes. However, these are not methods of business processes design – they do not provide any guidance on how to improve the quality attributes of designed systems like maintainability, performance, reusability etc. This is what the approach is aimed at.

In this paper we advocate an idea of transformational design of BPEL business processes in which specified behaviour remains preserved, while quality attributes get improved. There are three basic roots of our approach:

1. software refactoring – the approach introduced by Opdyke in [24], further developed in [20], in which the transformations

of source code are defined so as to improve its quality attributes;
2. business process design – in the realm of SOA informal or semiformal methods dominate the research carried out so far – comp. Service Responsibility and Interaction Design Method (SRI-DM) [21];
3. business process equivalence – there have already been developed several notions of the equivalence between business processes based on Petri Nets [19] and Process Algebras [29].

The transformations of Business Processes are in the core of our approach and represent similar concept as popular software refactorings. Our original notion of business process equivalence has been introduced on a formal Process Algebra model of business processes (explained and discussed in section *Behavioural Equivalence*) and it has been proved that the defined transformations create processes equivalent to the one being transformed. These transformed processes are compliant in terms of their behaviour, however, they have quality attributes changed. These transformations may be steered by the quality scenarios and assessments performed using Architecture Trade-off Analysis Method (ATAM) [18].

This provides a foundation for the transformational design method in which a starting BPEL process is subject to a series of transformations yielding as a result behaviourally compatible model with improved non-functional properties like modifiability, maintainability, performance, reusability etc.

2. State of the Art

Many business processes design and maintenance methods are based on *Business Process Management* (BPM) concepts [31]. According to BPM, process life-cycle consist of five phases:
1. design – existing business processes are analysed and "to-be" processed are designed. The results of this phase are: process flows, main actors, resources and so on;
2. modeling – the purpose of this part is to model and make conclusions on process execution before its practical application;
3. execution – in execution phase processes are put into practice and run in physical environment;
4. monitoring – running processes are monitored, functional and non-functional properties are measured;
5. optimization – this phase is responsible for improvement of processes.

The BPM concept is broadly applied in the processes domain, however, we believe that there is no specialised application in SOA context. Current paper tries to fill this gap.

In the field of general process modeling there are approaches based on *Unified Modeling Language* (UML) like presented in [28]. The authors present suitability of UML activity diagrams for business process.

In the context of Service Oriented Architecture there exist special methods devoted to design business processes like mentioned previously Service Responsibility and Interaction Design Method (SRI-DM) [21]. The SRI-DM method is based on transformation from UML use-cases towards services with proper divided functionality and sequence diagrams that express desired process.

The approach similar to proposed in the current paper is presented in [15]. The authors propose modeling business process as a Petri net and such transformations of the net to reach optimal value of some goal function. The proposed approach is based on optimization techniques.

The research of the current paper is concentrated on converting BPEL processes to one of the formal models that can be subject to model-checking techniques. A survey of such approaches can be found in [3]. It reveals that all of the most important formal models of concurrent systems have been applied: Petri nets (basic model, high-level, coloured) – comp. [14], [32], Process Algebras – comp. [12], [11], Lotos – comp. [9], [30], Promela and LTL – comp. [13], [16], Abstract State Machines – comp. [8], [27], Finite State Automata – comp. [11]. These conversions make it possible to detect deadlock and

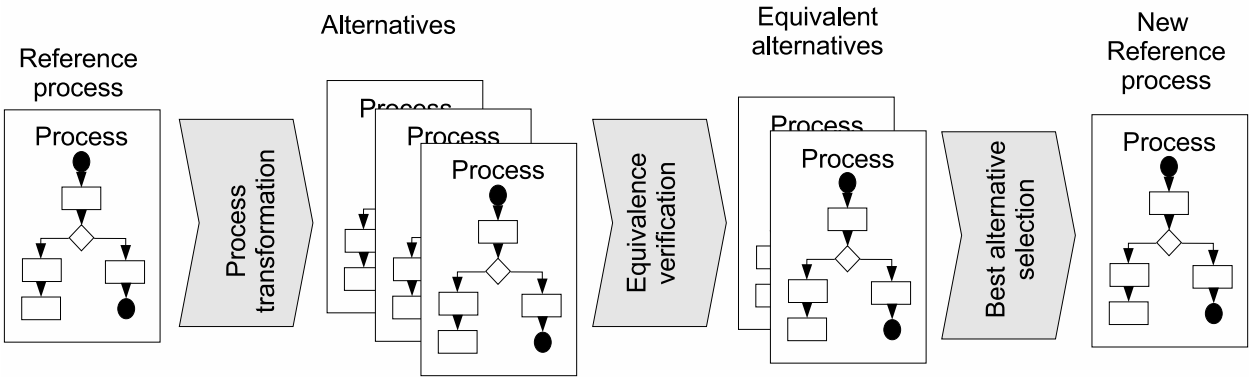


Figure 1. Process transformation algorithm

livelock as well as reachability analysis with automated model checkers.

3. Process Transformation Design Approach

The algorithm of process transformation design is depicted in Figure 1.

1. As it was mentioned in the introduction, the algorithm starts with the original process that is delivered by business oriented staff and the primal process bring up only functional aspects of the process. Functional aspects of the process are: necessary activities, order of activities, relation between them, exchanged data, basic external services invocation and so on. The process is called *reference process*. In following iterations the original process is slightly changed by refactoring transformations [25], [20] like:
   - service split – split one complex services into two or more smaller ones that cover the primary one functionality,
   - service aggregation – opposite to service split – composing two or more services in one larger service,
   - parallelization – making serial activities to run parallel,
   - asynchronization – reconstruction of communication protocol from synchronous to asynchronous.

   The above transformations are called *refactorings* and they are only examples of possible refactorings. Obviously, in a given process only some subset of transformations is possible and a smaller subset is rational.
2. A few independent refactorings on a current process make a few alternative processes which should be equivalent to the original process or at least changes in behaviour should be known.
3. Behaviour preservation is checked by means of behavioural equivalence verification step. In this step formal methods of Process Algebra (PA) [6] are used. The result of verification is either elimination of not-equivalent alternative or accepting changes in behaviour that the transformation makes. The way the transformation changes behaviour is exactly known owing to PA formalism.
4. After eliminating or accepting, all alternatives that are left are evaluated against interesting non-functional properties like:
   - performance,
   - safety,
   - maintainability,
   - availability,
   - or any important property.

   The measure of each property is calculated by using specified metrics, models [7] or simulations.
5. In the following step one alternative is selected amongst others. The selection is based on Architecture Trade-off Analysis Method (ATAM) [18]. In short, the method examines sensitivity of non-functional parameters to design properties and marks out *trade-off points*. Trade-off points are decision variables that affect more than one quality attributes. Changing the value of trade-off points in-

creases some quality attributes and decreases others. In case of SOA, services granulation is an example of such trade-off point. When services are bigger and not numerous then we have good performance and weak maintainability and reusability. If we split the system into more services, performance will decrease but maintainability and reusability will increase.

6. After selecting one alternative process, the selected process becomes new reference process and the algorithm returns to the beginning.

The above steps lead from process that is correct from functional point of view to process that has best or acceptable good non-functional quality attributes.

All steps of the algorithm are guided by a human designer and supported by automatic tools that may:

– suggest possible transformations of a reference process,
– verify behavioural equivalence,
– compute quality metrics of alternatives,
– point out trade-off points.

The conclusion is that the transformational approach does not try to make a totally automatic process design, because, in our opinion, it is impossible without human ability involvement. Instead, the transformational method supports a human designer's creative work in tasks difficult for a human.

4. Behavioural Equivalence

Behavioural equivalence verification is based on Process Algebra transformation and manipulation of BPEL processes [6].

4.1. Process Algebra for Behavioural Equivalence

Process Algebra (PA) [6] is formal semantic that express concurrent and distributed processing. It is specially devoted for parallel, loosely coupled and asynchronous communication so it is tailored to BPEL analysis. During our research we used LOTOS [2] realisation of PA.

Using the LOTOS notation, one can model any process or chain of communicating processes, simulate processes execution and, what is the most important in the context of refactoring, verify equivalence of two different processes. The equivalence is verified by *simulation*, *bisimulation* or *preordering* analysis [6].

To be able to use PA in stated problem it is necessary to use some kind of mapping from BPEL activities to PA terms. There are a few existing BPEL to PA mappings [10, 4], but none of them exactly fit to the needs of transformational process design. Firstly, because they demand full semantic checking in equivalence verification, that is too precise for refactoring. In case of the refactoring equivalence verification, if one process is transformed, its semantic changes but its behaviour does not. Another aspect is that an important property of mapping BPEL to PA for refactoring is that it has to make simple models with possibly the smallest statespace – during the design procedure there are a few changing scenarios and each of them has to be verified – the time spent for one verification is limited. This is the motivation for us to develop new mapping. Mappings of BPEL activities to PA formulas are presented in Table 1.

The mappings do not take into account data values or condition probability. This is motivated by simplification (and better verification performance) of the model. From another point of view, making some assumptions, there is no actual need to examine values of variables in equivalence verification.

There is an artificial mapping of activity which is not explicit part of BPEL but is necessary for equivalence verification. This is *activity dependency* mapping. Let us assume that there are two activities in BPEL process that are not directly attached to each other (by e.g. <sequence> or <switch>) but by shared variable, like in the following example:

```
<receive variable="PurchaseOrder"
         name="ReceivePurchase" />
...
<assign name="assignOrder">
  <copy>
```

Table 1. Sample mappings BPEL activities to PA formulas. Part 1

BPEL	LOTOS Process Algebra
empty <empty name="emptyName" [...] </empty>	process empty_emptyName[dummy] := exit endproc
external service invocation <invoke inputVariable="ivName" outputVariable="ovName" name="invName" [...]> [...] </invoke>	process invoke_invName[ivName,ovName] := ivName;ovName;exit endproc
receive message <receive variable="vName" name="receiveName" [...]> [...] </receive>	process receive_receiveName [vName] := vName;exit endproc
reply <reply variable="vName" name="replyName" [...] > [...] </reply>	process reply_replyName[vName] := vName;exit endproc
assign variable value <assign name="asgName" <copy> <from variable="fromVar"> <from to="toVar"> </copy> </assign>	process assign_asgName[fromVar, toVar] := fromVar;toVar;exit endproc
parallel execution <flow name="flowName"> < ... name="activityA"/> < ... name="activityB"/> [...] </flow>	process flow_flowName[dummy] := activityA || activityB ... endproc
sequential execution <sequence name="seqName"> < ... name="activityA"/> < ... name="activityB"/> [...] </sequence>	process sequence_seqName[linkSyn] := activityA >> linkSyn;activityB >> ... endproc Note: linkSyn should be placed according to potential link synchronization usage.

Table 2. Sample mappings BPEL activities to PA formulas. Part 2

BPEL	LOTOS Process Algebra
conditional execution <switch name="switchName"> <case ...> < ... name="activityA"/> </case> <case ...> < ... name="activityB"/> </case> </switch>	process switch_switchName[dummy] := hide ended in (activityA [ ] activityB ...) endproc
pick <pick name="pickName"> <onMessage partnerLink="ncname" portType="qname" operation="opA" variable="ncname"> activityA </onMessage> <onMessage partnerLink="ncname" portType="qname" operation="opB" variable="ncname"> activityB </onMessage> </pick>	process pick_pickName[dummy] := activityA[ ]activityB endproc
link <flow name="flowName"> <links> <link name="XtoY"/> </links> <sequence name="X"> <source linkName="XtoY"/> <invoke name="A" .../> <invoke name="B" .../> </sequence> <sequence name"Y"> <target linkName="XtoY"/> <invoke name="E" .../> </sequence> </flow	process flow_flowName[dummy] := hide XtoY in (sequence_X[XtoY] |[XtoY]| sequence_Y[XtoY]) endproc

```
    <from variable="PurchaseOrder"/>
    <to variable="ShippingRequest"/>
  </copy>
</assign>
```

Then activity dependency mapping will be:

```
process act_dependency[dummy]
  receive_ReceivePurchase[PurchaseOrder]
      |[PurchaseOrder]|
  assign_assignOrder[PurchaseOrder,
                     ShippingRequest]
endproc
```

The activity dependency expresses indirect dependency of two activities of which, one needs output data from another, no matter what structural dependency (sequence or parallel) in the process are.

4.2. BPEL Behavioural Equivalence

There are a few approaches to determine behavioural equivalence (or in other words behaviour preservation) of refactored processes. In [24] the author proposes such definition, that two systems are equivalent when the response for each request is the same from both systems. According to [22] communication-oriented systems are equivalent if they send messages in the same order.

In case of transformational design we assume that every service fulfills stateless postulate. It means that when BPEL process invokes external service then in every invocation response for some request is always the same, it is independent of history. This assumption leads to a conclusion that state of external services (and all environment) is encapsulated inside the invoking service.

To make this assumption usable and to prove how it can be used we needed some PA theory.

$$B \xrightarrow{x} B' \tag{1}$$

The above formula means that process B reaches state B' after receiving an event (message) x.

Now PA semantics is defined using *inference rules* that has form:

$$\frac{premises}{conclusions}(sidecondition) \tag{2}$$

For example parallel execution (without synchronization) || has 2 symmetric rules:

$$\frac{B1 \xrightarrow{x} B1'}{B1||B2 \xrightarrow{x} B1'||B2} \text{ and } \frac{B2 \xrightarrow{x} B2'}{B1||B2 \xrightarrow{x} B1||B2'} \tag{3}$$

an preceding (sequential composition) >> has 2 rules:

$$\frac{B1 \xrightarrow{x} B1'}{B1 >> B2 \xrightarrow{x} B1' >> B2} \text{ and } \frac{B2 \xrightarrow{\sigma} B2'}{B1 >> B2 \xrightarrow{i} B2} \tag{4}$$

where σ is successful termination and i is unobservable (hidden) event.

If external service S is stateless then:

$$\forall y \in Y S \xrightarrow{y} S \tag{5}$$

where Y is a set of all events. This means that every event, generated externally or from the subjected service, does not change the state and answer from the service.

To analyse a BPEL process using PA terms, the BPEL process has to be translated into PA using mapping mentioned in previous section. The product of translation is a set of PA processes that are sequentially ordered by BPEL steering instructions – sequences, flows, switches and so on. Additionally, a part of mapping is *activity dependency* processes. This artifact symbolizes data dependency between elements.

Let us symbolize it with *dependency operator*:

$$A]x]B \tag{6}$$

which means that state B can be started after A is successfully terminated and event x is emitted (or received).

Below we can see some example, that shows what is our behavioural equivalence based on.

The given process has a set of operations connected with dependency sequence:

$$(A]x]C]z]D) \tag{7}$$

C waits for A result and D for C result.

Beside the above dependency, the process has also structural sequence defined by <sequence> instruction $A \rightarrow B \rightarrow C \rightarrow D$, where B is instruction which is not connected by *activity dependency*. We can relax the structural sequence and consider the process as:

$$(A]x]C]z]D)||B \tag{8}$$

That means that we can treat $(A]x]C]z]D)$ and B as two parallel independent activities.

The proof that (8) is true for stateless services.

1. If there is no external service (8) is true by the definition because there is no interaction between $(A]x]C]z]D)$ and B,
2. If there is stateless external service S, then:

$$\forall y(A]x]C]z]D)||S \xrightarrow{y} ((A]x]C]z]D))'||S \tag{9}$$

and

$$\forall yS||B \xrightarrow{y} S||B' \tag{10}$$

which leads to:

$$\begin{gathered}(A]x]C]z]D) \xrightarrow{y} (A]x]C]z]D)' \\ \Rightarrow (A]x]C]z]D)||B \xrightarrow{y} (A]x]C]z]D)'||B\end{gathered} \tag{11}$$

and

$$\begin{gathered}B \xrightarrow{y} B' \\ \Rightarrow (A]x]C]z]D)||B \xrightarrow{y} (A]x]C]z]D)||B'\end{gathered} \tag{12}$$

The equation (12) is parallel execution inference rules (3) which is proof of (8)

If S was stateful, then

$$\exists y(A]x]C]z]D)||S \xrightarrow{y} (A]x]C]z]D)'||S' \tag{13}$$

then

$$(A]x]C]z]D)||B \xrightarrow{y} (A]x]C]z]D)'||B' \tag{14}$$

this would mean that there are some interactions between $(A]x]C]z]D)$ and B, and that they can not be treated independently.

The above theory makes it possible to divide the whole BPEL process into parts, that are only dependant by *activity dependency* and also makes possible to check if every refactored process is contained in these dependencies. This technique is related to *program slicing* [1] used broadly in source code refactoring. The BPEL service with defined activity dependencies and without structured constraints (sequences, flows, conditional and so on) is called *minimal dependency process* and is used to check the behavioural equivalence. After refactoring, the new (refactored) process has to be translated to PA and its PA image must fulfill preorder relationship with the *minimal dependency process*. Refactored process has to be subgraph of *minimal dependency process states graph*.

5. Transformation Steering

The process of transformations is steered by a method based on the Architecture Trade-off Analysis Methods (ATAM) [18]. The ATAM helps to identify *trade-off points*, that are parameters that have impact on a few quality aspects of the analyzed system. The impact of *trade-off points* is positive on one aspect and negative on another. So to designate proper value of such parameter there a trade-off has to be reach on this parameter.

ATAM helps to decide which alternative should be selected during the process design. In that way ATAM steers transformation in a design algorithm.

6. Process Design Example

In order to illustrate how transformational design works in practice, a simple example is presented below. The example is inspired by BPEL specification [17]. The quality of process is measured in two aspects: performance and reusability. The performance metric is response time under a given load, and reusability is measured by number of interfaces that whole service provides.

6.1. Reference process

The business process is a typical purchase of goods service. The service is composed of three activities: invoicing, order shipping and production scheduling. The activities of the process are organized as follows:

1. the process receives purchase order, receives product type, quantity and desired shipping method,
2. shipping service is requested and the price of shipping is received,
3. an invoice is requested from an invoicing service, the invoice contains product price and shipping price,
4. the production of goods is scheduled by request to a scheduling service.

Each activity is executed in sequence. Next activity starts after the previous is finished. The reference process and surrounding services are depicted in Fig. 2.

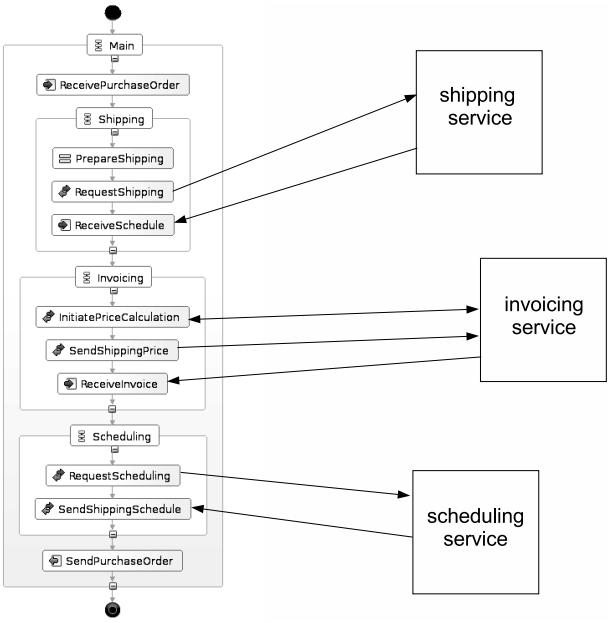


Figure 2. Purchase order reference process

6.2. Process Alternatives

For the current reference process, the designer proposes three alternatives that seem to be equivalent. Alternative (1) is a process that first makes request for shipping service and afterwards, parallelly requests shipping service and invoice service.

Alternative (2) starts all three requests parallelly – invoicing, shipping and scheduling service.

Alternative (3) is a bit more sophisticated – the reference service is split into three services. One of them invokes shipping service, the second one parallelly invokes invoicing and scheduling services, the third service composes two subservices. The alternatives are presented in Fig. 3.

6.3. Equivalence verification

In the current stage of algorithm, alternatives are verified to be behavioural equivalents to reference process. The technique of verification is described in section 5. The result of the verification is as follows:

– alternative 1 is behaviourally equivalent unconditionally,
– alternative 2 is not equivalent, because a request to invoicing service and shipping service depends on data received from shipping service. When all three requests startsat the same time, we can not guarantee, that the data from shipping service is received before a request to scheduling and invoicing services is made.
– alternative 3 is behaviourally equivalent.

Upon the above information, the designer decides to remove alternative 2 from the alternatives set.

6.4. Alternatives Evaluation – Performance

As it was mentioned at the beginning of the section, alternatives are evaluated in performance and reusability aspects. Performance is defined as a mean response time estimation. The web service and connections between services can be modeled, with queueing theory, as M/M/1//inf system. It means that requests arrive to the system independently with exponential interval distribution and response time is also exponentially distributed. Thanks to the above assumptions, average response time of whole system can be estimated as a sum of average responses from its components: services and links between them. To make evaluation simpler, we assume that every network connection has the same average latency R_N. So average response time of the reference process is:

$$R_{RP} = R_{BPEL_RP} + R_{\text{shipping}} + R_{\text{invoicing}} + R_{\text{scheduling}} + 7R_N \tag{15}$$

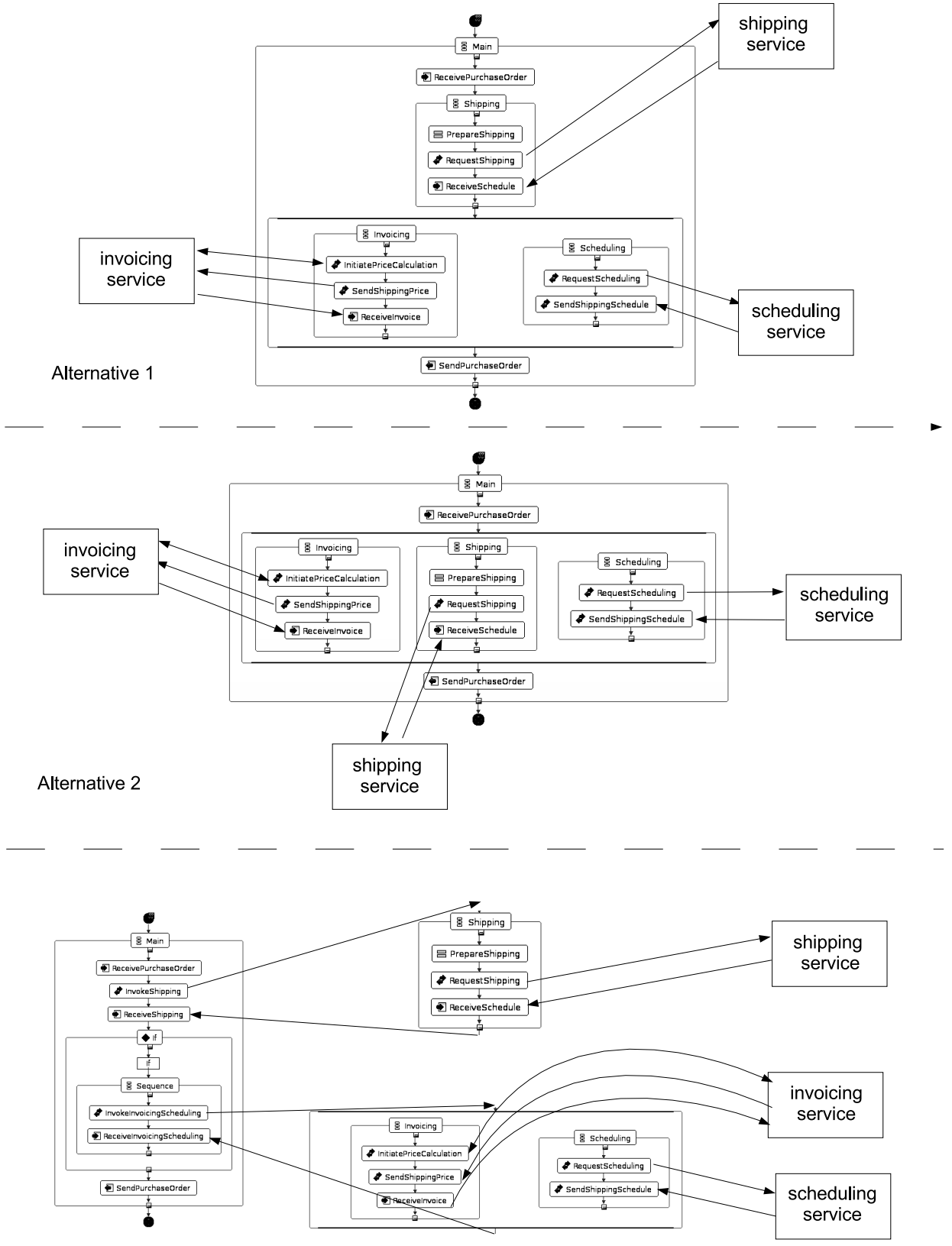


Figure 3. Possible alternatives for reference process

An important fact in the above equation, is that average response times of invoicing, shipping and scheduling are simply added, because requests to services are made consequently, one by one. Let us assume additionally values of each parameter:

- $R_{BPEL_RP} = 2$ ms (average time of processing of main BPEL process),
- $R_{\text{shipping}} = 3$ ms (avg. resp. time from shipping service),
- $R_{\text{invoicing}} = 5$ ms (avg. resp. time from invoicing service),
- $R_{\text{scheduling}} = 4$ ms (avg. resp. time from service),
- $R_N = 1$ ms (avg. network latency).

That gives $R_{RP} = 21$ ms.

For alternative 1 average response time is:

$$R_{A1} = R_{BPELA1} + R_{\text{shipping}} + \max(R_{\text{invoicing}}, R_{\text{scheduling}}) + 7R_N \quad (16)$$

the difference between alternative 1 and reference process is that invoice and scheduling services are requested parallelly, so response time from the parallel part is a maximum of response times from invoicing and scheduling. When we assume that $R_{BPELA1} = R_{BPELRP}$ then: $R_{A1} = 17$ ms.

Finally alternative 3 average response time is:

$$R_{A3} = R_{BPELA3_1} + R_{BPELA3_2} + R_{BPELA3_3} + R_{\text{shipping}} + \max(R_{\text{invoicing}}, R_{\text{scheduling}}) + 11R_N \quad (17)$$

that gives: $R_{A3} = 25$ ms.

6.5. Alternatives Evaluation – Reusability

As a reusability metrics is taken the total number of interfaces that a service delivers. Reference process and alternative 1 delivers four interfaces: one to main composed process and three to elementary services: invoicing, shipping and scheduling. Alternative 3 delivers 6 interfaces: three to basic services, one to composite service and two new interfaces to two sub services – shipping request and invoicing scheduling request.

All the above data are gathered in Table 3.

6.6. Best Alternative Selection

By means of ATAM method it is possible to identify the trade-off point, which is in following example services quantity. If composite service consist of more basic services, then it is more reusable, however, performance suffers.

In the current stage the new reference process has to be designated. Apparently alternative 1 is the best choice. Alternative 1 is better than the current reference process in performance measure and not worse in reusability. Alternative 3 is better in reusability than alternative 1 but much worse in performance, even worse than reference process.

7. Tool Support

As it was mentioned previously, an important goal of the research is to deliver a tool that will support usage of transformational process design. The tool is currently under development. In the current section a current status of tool development is described. The tool is based on open-source NetBeans IDE [23]. It is planned that whole design process will be held in NetBeans. BPEL editor, which is already implemented in the IDE, is used. Beside BPEL editor, a graphical editor is necessary as it will guide

Table 3. Quality metrics for reference process and alternatives

	Reference process	Alternative 1	Alternative 3
Average response time	21 ms	17 ms	25 ms
Reusability	4	4	6
Services quantity	1	1	3

the process design iteration – its layout will be similar to Figure 1. The editor will be the main window of the tool. A designing user will be able to click on every alternative and look inside using native BPEL editor. In the main window there will also be all the important data about quality of alternatives.

7.1. BPEL Refactoring

To automate refactoring process in BPEL language it was necessary to create the tool which provides these features. It was proposed to automate such types of transformation: renaming (variable, partnerLink, and correlationSet), aggregation, asynchronization, parallelization, split. After selecting a part of the code in BPEL file one of the mentioned transformations can be realized (if it is possible). Such tool has not been already implemented – this is why I decided to implement an idea of creating the application as a plug-in to Netbeans IDE which automates refactoring process. There are numerous engineering challenges connected with the detailed design of tool support for BPEL transformations. These have been presented in detail below.

7.1.1. Renaming

It is the simplest type of refactoring – changes the name one of the three elements in BPEL (variable, partnerLink, and correlationSet) and all the occurrences of this element in other language constructions. It seems to be an easy transformation but it is relevant. It would be difficult to do it manually because BPEL contains a lot of constructions with reference to other elements. For instance reference to variable may occur in such elements: receive, reply, invoke, onMessage, throw, copy from, copy to and in XPath expressions: wait, onAlarm, if, else if, while, repeatUntil, forEach. As we can see it is much easier to use an automatic tool which finds all the occurrences of the chosen element in BPEL code. The offered application provides these features. We can change the name one of the mentioned elements and do not have to worry about occurrences in other BPEL constructions – program will do it for us automatically.

7.1.2. Aggregation

Composing one or more services into larger one seems to be easy. It is because somebody may think that it is enough to move logic from one service to another and that is all. It is a wrong approach because there are a lot of other elements which we have to focus on.

First of all, we must find the BPEL file that contains the logic of the invoked process which is automatically done by the proposed tool.

Secondly, it is needed to move elements such as variable, partnerLink, correlationSet and namespaces to the process that is invoking, because all the elements are used in logic which we want to encapsulate.

Last but not least, it may happen that the used variables, partnerLinks or namespaces in invoked process have the same name as in process which is invoking the first one. This situation is considered in the proposed tool – when the situation occurs, application changes the name of the specified element in all constructions where reference to this element occur in order to prevent name collision. A similar situation may happen in namespaces because the one we want to add is already defined. In this case it is also needed to change the name of the added namespace in every place where it occurs. Also a very important thing is to ensure that variable used as input in invoked process (attribute variable in receive element) after the transformation will be the same as input in invoke element before transformation. A similar situation occurs when we have synchronous invocation with output variable it has to be checked whether variable used in reply element will be the same as an output variable in invoke element before refactoring. This situation is also supported by the application.

7.1.3. Asynchronization

In this type of refactoring the offered tool also provides a few conveniences that automate process of transformation. First of them is finding

as many operations as it is possible which are invoked after selected element and they are independent. After that we can change the invocation method from synchronous to asynchronous. If there are no independent operations transformation will be terminated.

To change the invocation from synchronous to asynchronous some changes in WSDL and BPEL file in invoked process are needed. We have to delete (in WSDL file) an output element in operation construction (to make invocation asynchronous) and add a new input element for a reply to the primary process (we can not use the same input element for the reply because the types of used variables may be different). Moreover in BPEL file we must change synchronous element reply to element invoke to make connection asynchronous – we need to define additional partnerLink element to make the connection possible. The application supports all of these transformations.

To finish the transformations it is necessary to provide some modifications in the primary process. This is because of the type of invocation (asynchronous) which we introduce earlier by changing a partner WSDL file. After all independent operations we need to place element receive to collect a response from partner process and delete an attribute outputVariable in the invoke element.

The last thing to remember is to define correlation element to ensure that response will be transferred to the right instance of the primary process. This is why proposed tool makes some modifications in WSDL file of partner process. To be more accurate application defines property element and two propertyAlias elements. Thanks to that it is possible to define correlationSet and correlation elements in primary process file which guarantee that message will be delivered to right process instance. After all mentioned operations, which proposed tool supports, refactoring is finished.

7.1.4. Split

Splitting the service without using the automatic tool may also be difficult. To extract a part of the service and then create another service to be invoked inside the primary service we have to create two new files – a BPEL file and a WSDL file. Moreover, we must fill them with all the necessary information which is indispensable to make a network connection with the new process. As well it is requisite to change the primary process so that the connection with the new process will be possible. All the mentioned operations are supported by our tool.

First of all, the application chooses two variables – one as input variable and second as output variable for synchronous invocation of the new process. Choosing variables is not complicated operation because as input variable is chosen first which occurs in selected code to extract and it is used for one of the operations. In case of the occurrence more than one variable, all of which are not initiated in a chosen logic, the transformation will be terminated. Selection of the output variable is very similar to the selection of the input variable – if exists exactly one variable, which is initiated in the selected logic and used later after the selected code, it will be chosen as output variable.

Next, BPEL and WSDL files are created. In a WSDL file all the necessary constructions are created, such as: message, portType, operation, partnerLinkType and namespaces which defines complex types of the variables. Then using definition created in a WSDL file it is possible to make a new BPEL file and create constructions: variable, partnerLink, namespaces, etc. and place selected logic in new file. All of the operations are supported by the application.

At the end it is necessary to modify the primary process. To make a connection with the new process the application adds invoke activity (with all attributes) instead of the extracted logic and element partnerLink (also with all attributes). After all these transformations splitting the process into parts is possible.

7.2. Tools for Equivalence Verification

The algorithm of equivalence verification consist of three steps:

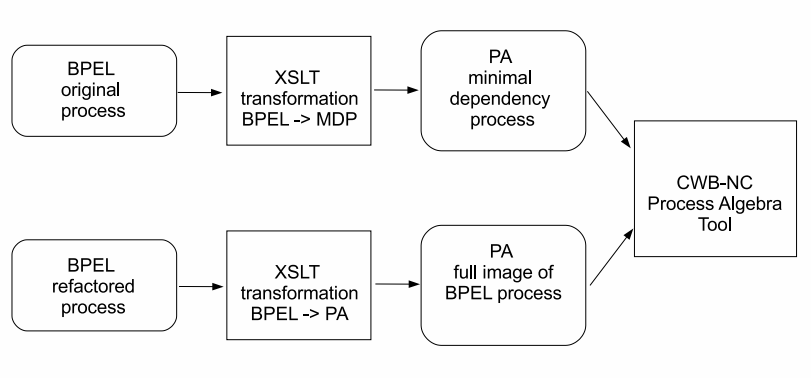


Figure 4. Structure of verification process

1. translating BPEL process to minimal dependency process (MDP) – this step is made only once at the beginning of the refactoring process,
2. translating BPEL process to its PA image,
3. checking preorder relationship of PA image with minimal dependency process.

As a part of the developed tool, translation BPEL to PA was made by means of an XSLT processor, as the PA processor was used Concurrence Workbench for New Century (CWB-NC) [5]. The structure of verification system is in the Figure 4.

The transformation from BPEL to its PA image is a quite trivial action as it was used XSLT preprocessor. The XSLT processor automatically maps BPEL instructions into their PA equivalences as it is listed in Tables 1 and 2.

The second type of mapping – from BPEL to its MDP image is more sophisticated. As it is needed to resolve indirect dependencies between BPEL activities there graph manipulation techniques are applied.

8. Summary and Further Work

A method for transformational design of SOA business processes in BPEL has been presented. It has been founded on the formal framework of process algebras as well as the concept of process equivalence originally developed by the authors. The transformations are aimed at improving certain properties like e.g. modifiability and performance while preserving the behaviour specified by the starting business process model. The whole framework has been accompanied by a prototype tool which has been integrated with NetBeans environment in the form of a plug-in. The challenges resolved during tool development have by no means turned out to be trivial. Therefore, they have also been discussed in the paper which should become a valuable resource of the real implementation experiences in the field of transforming BPEL as well as for the continuation of the work presented here.

The approach has been validated on an exemplary design case. The result of such a case study are promising though some more complicated cases would provide a chance for a more in-depth validation of the whole approach.

References

[1] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[3] F. Breugel and M. Koshkina. Models and verification of BPEL. 2006.

[4] J. Cámara, C. Canal, J. Cubo, and A. Vallecillo. Formalizing WSBPEL business processes using process algebra. *Electr. Notes Theor. Comput. Sci.*, 154(1):159–173, 2006.

[5] R. Cleaveland. Concurrency workbench of the new century, 2000. http://www.cs.sunysb.edu/~cwb/.

[6] R. Cleaveland and S. Smolka. Process algebra. 1999.

[7] A. D'Ambrogio and P. Bocciarelli. A model-driven approach to describe and predict the performance of composite services. In *WOSP '07: Proceedings of the 6th international*

workshop on Software and performance, pages 78–89, New York, NY, USA, 2007. ACM.

[8] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative control flow. In *Abstract State Machines*, pages 131–152, 2005.

[9] A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.

[10] A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.

[11] H. Foster, J. Kramer, J. Magee, and S. Uchitel. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.

[12] H. Foster, S. Uchitel, J. Magee, J. Kramer, and M. Hu. Using a rigorous approach for engineering web service compositions: A case study. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 217–224, Washington, DC, USA, 2005. IEEE Computer Society.

[13] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.

[14] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proceedings of the BPM 2005*, pages 220–235, Nancy, France, Sept. 2005. Springer-Verlag.

[15] I. Hofacker and R. Vetschera. Algorithmical approaches to business process design. *Computers & OR*, 28(13):1253–1275, 2001.

[16] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[17] IBM, BEA, and Microsoft. Business process execution language for web services. http://citeseer.ist.psu.edu/669609.html, 2003.

[18] R. Kazman, M. H. Klein, M. Barbacci, T. A. Longstaff, H. F. Lipson, and S. J. Carrière. The architecture tradeoff analysis method. In *Proceedings of ICECCS*, pages 68–78, 1998.

[19] A. Martens. Simulation and equivalence between BPEL process models. In *Proc. of Intl. Conference DASD'05*, 2005.

[20] F. Martin. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] D. E. Millard, H. C. Davis, Y. Howard, L. Gilbert, R. J. Walters, N. Abbas, and G. B. Wills. The service responsibility and interaction design method: Using an agile approach for web service design. pages 235–244, 2007.

[22] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 235–250, New York, NY, USA, 1996. ACM.

[23] NetBeans IDE. http://www.netbeans.org/.

[24] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[25] A. Ratkowski and A. Zalewski. Performance refactoring for service oriented architecture. *ISAT '2007: Information Systems Architecture And Technology*, 2007.

[26] A. Ratkowski and A. Zalewski. Transformational design of business processes in SOA. In *Proceedings of CEE-SET*, 2008.

[27] W. Reisig. Modeling and Analysis Techniques for Web Services and Business Processes. In *FMOODS 2005, Athens, Greece, June 15–17, 2005. Proceedings*, volume 3535, pages 243–258. Springer Verlag, May 2005.

[28] N. Russell, W. van der Aalst, Arthur, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *APCCM '06: Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling*, pages 95–104, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[29] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 43, Washington, DC, USA, 2004. IEEE Computer Society.

[30] G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among web services using LOTOS/-CADP. In *ECOWS*, pages 198–212, 2004.

[31] W. van der Aalst, A. ter Hofstede, and M. Weske. Business process management: A survey. In *Business Process Management*, *Lecture Notes in Computer Science*, pages 1–12. Springer, Berlin, Heidelberg, 2003.

[32] Y. Yang, T. Tan, J. Yu, and F. Liu. Transformation BPEL to CP-Nets for verifying web services composition. In *Proceedings of NWESP '05*, page 137, Washington, DC, USA, 2005. IEEE Computer Society.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Satisfying Stakeholders' Needs – Balancing Agile and Formal Usability Test Results

Jeff Winter*, Kari Rönkkö*
*School of Engineering, Dept. of Interaction & System Design, Blekinge Institute of Technology
jeff.winter@bth.se, kari.ronkko@bth.se

Abstract

This paper deals with a case study of testing with a usability testing package (UTUM), which is also a tool for quality assurance, developed in cooperation between industry and research. It shows that within the studied company, there is a need to balance agility and formalism when producing and presenting results of usability testing to groups who we have called Designers and Product Owners. We have found that these groups have different needs, which can be placed on opposite sides of a scale, based on the agile manifesto. This becomes a Designer and a Product Owner Manifesto. The test package is seen as a successful hybrid method combining agility with formalism, satisfying organisational needs, and fulfilling the desire to create a closer relation between industry and research.

1. Introduction

Product quality is becoming the dominant success criterion in the software industry, and Osterweil states that the challenge for research is to provide the industry with the means to deploy quality software, allowing companies to compete effectively [23]. Quality is multi-dimensional, and impossible to show through one simple measure, and research should focus on identifying various dimensions of quality and measures appropriate for it [23]. A more effective collaboration between practitioners and researchers would be of great value [23]. Quality is also important owing to the criticality of software systems (a view supported by Harrold in her roadmap for testing [14]) and even to changes in legislation that make executives responsible for damages caused by faulty software.

One approach to achieving quality has been to rely on complete, testable and consistent requirements, traceability to design, code and test cases, and heavyweight documentation. However, a demand for continuous and rapid results in a world of continuously changing business decisions often makes this approach impractical or impossible, pointing to a need for agility. At a keynote speech at the 5th Workshop on Software Quality, held at ICSE 2007 [45], Boehm stated that both agility and quality are becoming more and more important. Many areas of technology exhibit a tremendous pace of change, due to changes in technology and related infrastructures, the dynamics of the marketplace and competition, and organisational change. This is particularly obvious in mobile phone development, where their pace of development and penetration into the market has exploded over the last 5 years. This kind of situation demands an agile approach [6].

This article is based on two case studies of a usability evaluation framework called UIQ Technology Usability Metrics (UTUM) [39], the result of a long research cooperation between the research group "Use-Oriented Design and Development" (U-ODD) [37] at Blekinge Institute of Technology (BTH), and UIQ Technology (UIQ) [38]. With the help of Martin et al.'s study [21]

and our own case studies, it presents an approach to achieving quality, related to an organizational need for agile and formal usability test results. We use concepts such as "agility understood as good organizational reasons" and "plan driven processes as the formal side in testing", to identify and exemplify a practical solution to assuring quality through an agile approach. The research question for the first case study was:

– How can we balance demands for agile results with demands for formal results when performing usability testing for quality assurance?

We use the term "formal" as a contrast to the term "agile" not because we see agile processes as being informal or unstructured, but since "formal" is more representative than "plan driven" to characterise the results of testing and how they are presented to certain stakeholders. We examine how the results of the UTUM test are suitable for use in an agile process. eXtreme Programming (XP) is used as an illustrative example in this article, but note that there is no strong connection to any particular agile methodology; rather, there is a philosophical connection between the test and the ideas behind the agile movement. We examine how the test satisfies requirements for formal and informal statements of usability and quality.

In the first study, we identify two groups of stakeholders that we designated as Designers (D) and Product Owners (PO), with an interest in the different elements of the test data. A further case study was performed to discover if these findings could be confirmed. It attempted to answer the following research questions:

– Are there any presentation methods that are generally preferred?
– Is it possible to find factors in the data that allow us to identify differences between the separate groups (D & PO) that were tentatively identified in the case study presented in the previous chapter?
– Are there methods that the respondents think are lacking in the presentation methods currently in use within UTUM?
– Do the information needs, and preferred methods change during different phases of a design and development project?
– Can results be presented in a meaningful way without the test leader being present?

The structure of the article is as follows. An overview of two testing paradigms is provided. A description of the test method comes next, followed by a presentation of the methodology, and the material from the case studies, examining the balance between agility and formalism, the information needs of different stakeholders, the relationship between agility, formality and quality, and the need for research/industry cooperation. The article ends with a discussion of the work, and conclusions.

2. Testing – Prevailing Models vs. Agile Testing

Testing is performed to support quality assurance, and an emphasis on software quality requires improved testing methodologies that can be used by practitioners to test their software [14]. Since we regard the test framework as an agile testing methodology, this section presents a discussion of testing from the viewpoints of both the software engineering community and the agile community.

Within software engineering, there are many types of testing, in many process models, (e.g. the Waterfall model [30], Boehm's Spiral model [4]). Testing is often phase based, and the typical stages of testing (see e.g. [33], [25]) are *Unit testing, Integration testing, Function testing, Performance testing, Acceptance testing, and Installation testing.* The stages from Function testing and onwards are characterised as *System Testing*, where the system is tested as a whole rather than as individual pieces [25]. Usability testing (otherwise named Human Factors Testing) has been characterised as investigating requirements dealing with the user interface, and has been regarded as a part of Performance testing [25]. The prevailing approach to testing is reliant on formal aspects and best practice.

Agile software development changes how software development organisations work, especially regarding testing [34]. In agile development, exemplified here by XP [1], a key tenet is that testing is performed continuously by developers. Tests should be isolated, i.e. should not interact with the other tests that are written, and should preferably be automatic, although not all companies applying XP automate all tests [21]. Tests come from both programmers and customers, who create tests that serve to increase their confidence in the operation of the program. Customers specify functional tests to show that the system works how they expect it to, and developers write unit tests to ensure that the programs work how they think it does. These are the main testing methods in XP, but can be complemented by other types of tests when necessary. Some XP teams may have dedicated testers, who help customers translate their test needs into tests, who can help customers create tools to write, run and maintain their own tests, and who translate the customer's testing ideas into automatic, isolated tests [1].

The role of the tester is a matter of debate. It is primarily developers who design and perform testing. However, within industry, there are seen to be fundamental differences between the people who are "good" testers and those who are good developers. In theory, it is often assumed that the tester is also a developer, even when teams use dedicated testers. Within industry, however, it is common that the roles are clearly separated, and that testers are generalists with the kind of knowledge that users have, who complement the perspectives and skills of the testers. A good tester can have traits that are in direct contrast with the traits that good developers need (see e.g. Pettichord [24] for a discussion regarding this). Pettichord claims that good testers think empirically in terms of observed behaviour, and must be encouraged to understand customers' needs. Thus, although there are similarities, there are substantial differences in testing paradigms, how they treat testing, and the role of the tester and test designer. In our testing, the test leaders are specialists in the area of usability and testing, and generalists in the area of the product and process as a whole.

3. The UTUM Usability Evaluation Framework

UTUM is a usability evaluation framework for mass market mobile devices, and is a tool for quality assurance, measuring usability empirically on the basis of metrics for satisfaction, efficiency and effectiveness, complemented by a test leader's observations. Its primary aim is to measure usability, based on the definition in ISO 9241-11, where usability is defined as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [17]. This is similar to the definition of quality in use defined in ISO 9126-1, where usability is instead defined as understandability, learnability and operability [18]. The intention of the test is also to measure "The User eXperience" (UX), which is seen as more encompassing than the view of usability that is contained in e.g. the ISO standards [39], although it is still uncertain how UX differs from the traditional usability perspective [41] and exactly how UX should be defined (for some definitions, see e.g. ([15], [16], [42]).

In UTUM testing, one or more test leaders carry out the test according to predefined requirements and procedure. The test itself takes place in a neutral environment rather than a lab, in order to put the test participant at ease. The test is led by a test leader, and it is performed together with one tester at a time. The test leader welcomes the tester, and the process begins with the collection of some data regarding the tester and his or her current phone and typical phone use. Whilst the test leader is preparing the test, the tester has the opportunity to get acquainted with the device to be tested, and after a few minutes is asked to fill in a hardware evaluation, a questionnaire regarding attitudes to the look and feel of the device.

The tester performs a number of use cases on the device, based on the tester's normal phone

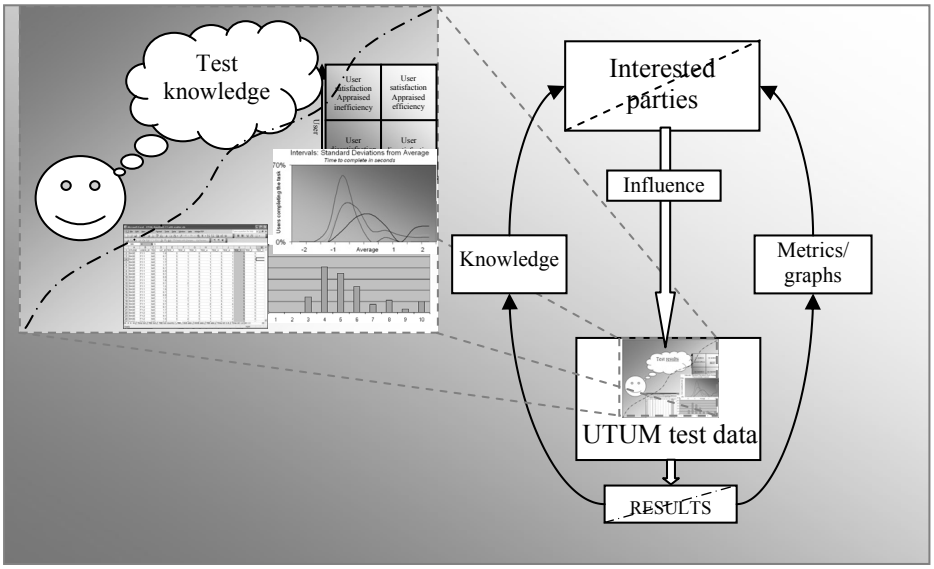


Figure 1. Contents of the UTUM testing, a mix of metrics and mental data

use or organisational testing needs. The test leader observes what happens during the use case performance, and records any observations, the time taken to complete the use cases, and answers to follow-up questions that arise. After the use case is complete, the tester answers questions about how well the telephone lets the user accomplish the use case.

When all of the use cases are completed, the tester completes a questionnaire based on the System Usability Scale (SUS) [7] about his or her subjective impressions of how easy the interface is to use. It expresses the tester's opinion of the phone as a whole. The tester is finally thanked for their participation in the test, and is usually given a small gift, such as a cinema ticket, to thank them for their help.

The data obtained are transferred to spreadsheets. These contain both quantitative data, such as use case completion times and attitude assessments, and qualitative data, such as comments made by testers and information about problems that arose. The data is used to calculate metrics for performance, efficiency, effectiveness and satisfaction, and the relationships between them, leading to a view of usability for the device as a whole. The test leader is an important source of data and information in this process, as he or she has detailed knowledge of what happened during testing.

Figure 1 illustrates the flow of data and knowledge contained in the test and the test results, and how the test is related to different groups of stakeholders. Stakeholders, who can be within the organisation, or licensees, or customers in other organisations, can be seen at the top of the flow, as interested parties. Their requirements influence the design and contents of the test. The data collected is found both as knowledge stored in the mind of the test leader, and as metrics and qualitative data in spreadsheets.

The results of the testing are thereby a combination of metrics and knowledge, where the different types of data confirm one another. Metrics based material is presented in the form of diagrams, graphs and charts, showing comparisons, relations and tendencies. This can be corroborated by the knowledge possessed by the test leader, who has interacted with the testers and who knows most about the process and context of the testing. Knowledge material is often presented verbally, but can if necessary be supported and confirmed by metrics and visual presentations of the data.

UTUM has been found to be a customer driven tool that is quick and efficient, is easily transferable to new environments, and that handles complexity [44]. For more detailed information on the contents and performance of

the UTUM test and the principles behind it, see ([39], and [44]). A brief video presentation of the whole test process (6 minutes) can be found on YouTube [8].

4. The Study Methodology and the Case Studies

This work has been part of a long-term research cooperation between U-ODD and UIQ, which has centred on the development and evaluation of a usability evaluation framework (for more information, see [44], [40]). The case studies in this phase of the research cooperation were based on tests performed by together by UIQ in Ronneby, and by Sony Ericsson Mobile Development in Manchester.

The process of research cooperation is action research (AR) according to the research and method development methodology called Cooperative Method Development (CMD), see [11], [10], [12] and ([28], chapter 8) for further details. AR "involves practical problem solving which has theoretical relevance" ([22] p. 12). It involves gaining an understanding of a problem, generating and spreading practical improvement ideas, applying the ideas in a real world situation and spreading the theoretical conclusions within academia [22]. Improvement and involvement are central to AR, and its purpose is to influence or change some aspect of whatever the research has as its focus ([27] p. 215). A central aspect of AR is collaboration between researchers and those who are the focus of the research. It is often called participatory research or participatory action research ([27] p. 216). CMD is built upon guidelines that include the use of ethnomethodological and ethnographically inspired empirical research, combined with other methods if suitable. Ethnography is a research strategy taken from sociology, with foundations in anthropology [29]. It relies upon the first-hand experience of a field worker who is directly involved in the setting that is under investigation [29]. CMD focuses on shop floor development practices, taking the practitioners' perspective when evaluating the empirical research and deliberating improvements, and involving the practitioners in the improvements. This approach is inspired by a participatory design (PD) perspective. PD is an approach towards system design in which those who are expected to use the system are actively involved and play a critical role in its design. It includes stakeholders in design processes, and demands shared responsibility, participation, and a partnership between users and implementers [32].

These studies have been performed as case studies, defined by Yin as "an empirical enquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" ([46], p. 13). Yin presents a number of criteria that are used to establish the quality of empirical social research and states that they should be applied both in the design and conduct of a case study. They deal with construct validity, internal validity, external validity and reliability ([46], pp. 35–39).

Three tactics are available to increase construct validity, which deals with establishing correct measures for the concepts being studied, and is especially problematic in case study research. These are: using multiple sources of information; ensuring a chain of evidence and; using member checking, i.e. having the key participants review the case study report. In this study, we have used many different sources of information. The data was obtained through observation, through a series of unstructured and semi-structured interviews [27], both face-to-face and via telephone, through participation in meetings between different stakeholders in the process, and from project documents and working material. The interviews have been performed with test leaders, and with staff on management level within the two companies. Interviews have been audio taped, and transcribed, and all material has been stored. The second case study involves the use of a survey. The mix of data and collection methods has given a triangulation of data that serves to validate the results that have been reached.

To ensure a chain of evidence a "study database" or research diary has been main-

tained. It collects all of the information in the study, allowing for traceability and transparency of the material, and reliability [46]. It is mainly computer based, and is an account of the study recording activities performed in the study, transcriptions of interviews and observation notes, and records of relevant documents and articles. The audio recordings are also stored digitally. The written document contains notations of thoughts concerning themes and concepts that arise when reading or writing material in the account of the study. The chain of evidence is also a part of the writing process.

The most important research collaborators in the industrial organisation have been an integral part of the study, and have been closely involved in many stages of the work. They have been available for testing thoughts and hypotheses during the study, giving opportunities for member checking. They have also been involved as co-authors when writing articles, which also means that member checking has been an integral part of the research.

Internal validity is especially important in exploratory case studies, where an investigator tries to determine whether one event leads to another. It must be possible to show that these events are causal, and that no other events have caused the change. If this is not done, then the study does not deal with threats to internal validity. Some ways of dealing with this are via pattern matching, explanation building, addressing rival explanations, and using logic models. This study has been a mix of exploratory and explanatory studies. To address the issues of internal validity in the case studies, we have used the general repertoire of data analysis as mentioned in the previous paragraph. The material in the research diary has been analysed to find emerging themes, in an editing approach that is consistent with Grounded Theory (see Robson [27] p. 458). The analysis process has affected the further rounds of questioning, narrowing down the focus, and shifting the main area of interest, opening up for the inclusion of new respondents who shed light on new aspects of the study. A further method for ensuring validity has been through discussions together with research colleagues, giving them the chance to react to the analysis and suggest and discuss alternative explanations or approaches.

External validity, knowing whether the results of a case study are generalisable outside the immediate case study, has been seen as a major hinder to doing case studies, as single case studies have been seen as a poor basis for generalisation. However, this is based on a fallacious analogy, where critics contrast the situation to survey research, where samples readily generalise to a larger population. In a case study, the investigator tries to generalise a set of results to a wider theory, but, generalisation is not automatic, and a theory must be tested by replicating the findings, in much the same way as experiments are replicated. Although Yin advises performing multiple-case studies, since the chances of doing a good case study are better than using a single-case design ([46], p. 53), this study has been performed as a single-case study and has been performed to generate theory. The case here represents a unique case ([46], p. 40), since the testing has mainly been performed within UIQ, and it is thereby the only place where it has been possible to evaluate the testing methodology in its actual context. One particular threat is in our study is therefore that most of the data comes from UIQ. Due to close proximity to UIQ, the interaction there has been frequent and informal, and everyday contacts and discussions on many topics have influenced the interviews and their analysis. Interaction with Sony Ericsson has been limited to interviews and discussions, but data from Sony Ericsson confirms what was found at UIQ. A further threat is that most of the data in the case study comes from informants who work within the usability/testing area, but once again, they come from two different organisations and corroborate one another, have been complemented by information from other stakeholders, and thus present a valid picture of industrial reality.

A threat in the second case study is the fact that only ten people have participated. This makes it difficult to draw generalisable conclusions from the results. Also, since the company is now disbanded, it is not possible to return

to the field to perform cross checking with the participants in the study. The analysis is therefore based on the knowledge we have of the conditions at the company and the context where they worked, and is supported by discussions with a people who were previously employed within the company, whom we are still in contact with. These people can however mainly be characterised as Designers, and therefore may not accurately reflect the views of Product Owners.

Thus, since this research is mainly based on a study of one company in a limited context, it is not possible to make confident claims about the external validity of the study. However, we can say that we have created theory from the study, and that readings appear to suggest that much of what we have found in this study can also be found in other similar contexts. Further work remains to see how applicable the theory is for other organisations in other or wider contexts. Extending the case study and performing a similar study in another organisation is a way of testing this theory, and further analysis may show that the case at UIQ is actually representative of the situation in other organisations.

Reliability deals with the replicability of a study, whereby a later investigator should be able to follow the same procedures as a previous investigator, and arrive at the same findings and conclusions. By ensuring reliability you minimize errors and bias in a study. One prerequisite for this is to document procedures followed in your work, and this can be done by maintaining a case study protocol to deal with the documentation problem, or the development of a case study database. The general way to ensure reliability is to conduct the study so that someone else could repeat the procedures and arrive at the same result ([46], pp. 35–39). The case study protocol is intended to guide the investigator in carrying out the data collection. It contains both the instrument and the procedures and general rules for data collection. It should contain an overview of the project, the field procedures, case study questions, and a guide for the case study report ([46], p. 69). As mentioned previously, a case study database has been maintained, containing the most important details of the data collection and analysis process. This ensures that the study is theoretically replicable. One problem regarding the replicability of this study, however, is that the rapidly changing conditions for the branch that we have studied mean that the context is constantly changing, whereby it is difficult to replicate the exact context of the study.

In the following, we begin by presenting the results of the first case study, and discuss in which way the results are agile or plan-driven/formal, who is interested in the different types of results, and which of the organisational stakeholders needs agile or formal results.

5. Agile or Formal?

The first focus of the study was the fact that testing was distributed, and the effect this had on the testing and the analysis of the results During the case study, as often happens in case studies [46], the research question changed. Gradually, another area of interest became the elements of agility in the test, and the balance between the formal and informal parts of the testing. The framework has always been regarded as a tool for quality, and verifying this was one purpose of the testing that this case study was based on. Given the need for agility mentioned above, the intention became to see how the test is related to agile processes and whether the items in the agile manifesto can be identified in the results from the test framework. The following is the result of having studied the material from the case study from the perspective of the spectrum of different items that are taken up in the agile manifesto.

The agile movement is based on core values, described in the agile manifesto [35], and explicated in the agile principles [36]. The agile manifesto states that: "We are uncovering better ways of developing software by doing it and by helping others do it. Through this work we have come to value: *Individuals and interactions* over processes and tools, *Working software* over comprehensive documentation, *Customer collaboration* over contract negotiation, and *Responding*

to change over following a plan. That is, while there is value in the items on the right, we value the items on the left more". Cockburn stresses that the intention is not to demolish the house of software development, represented here by the items on the right (e.g. working software over *comprehensive documentation*), but claims that those who embrace the items on the left rather than those on the right are more likely to succeed in the long run [9]. Even within the agile community there is some disagreement about the choices, but it is accepted that discussions can lead to constructive criticism. Our analysis showed that all these elements could be identified in the test and its results.

In our research we have always been conscious of a division of roles within the company, often expressed as "shop floor" and "management", and working with a participatory design perspective we have worked very much from the shop floor point of view. During the study, this viewpoint of separate groups emerged and crystallised, and two disparate groups became apparent. We called these groups Designers, represented by e.g. interaction designers and system and interaction architects, representing the shop floor perspective, and Product Owners, including management, product planning, and marketing, representing the management perspective.

When regarding this in light of the Agile manifesto, we began to see how different groups may have an interest in different factors of the framework and the results that it can produce, and it became a point of interest to see how these factors related to the manifesto and which of the groups, Designers (D) or Product Owners (PO), is mainly interested in each particular item in the manifesto. The case study data was analysed on the basis of these emerging thoughts. Where the groups were found to fit on the scale is marked in bold text in the paragraphs that follow. One of the items is changed from "Working software" to "Working information" as we see the information resulting from the testing process as a metaphor for the software that is produced in software development.

- **Individuals and interactions** – The testing process is dependent on the individuals who lead the test, and who actually perform the testing on the devices. The central figure here is the test leader, who functions as a pivot point in the whole process, interacting with the testers, observing and registering the data, and presenting the results. This interaction is clearly important in the long run from a PO perspective, but it is **D** who has the greatest and immediate benefit of the interaction, showing how users reacted to design decisions, that is a central part of the testing.
- **Processes and Tools** – The test is based upon a well-defined process that can be repeated to collect similar data that can be compared over a period of time. This is important for the designers, but in the short term they are more concerned with the everyday activities of design and development that they are involved in. Therefore we see this as being of greatest interest to **PO**, who can get a long-term view of the product, its development, and e.g. comparisons with competitors, based on a stable and standardised method.
- **Working information** – The test produces working information quickly. Directly after the short period of testing that is the subject of this case study, before the data was collated in the spreadsheets, the test leaders met and discussed and agreed upon their findings. They could present the most important qualitative findings to system and interaction architects within the two organisations 14 days after the testing began, and changes in the implementation were requested soon after that. An advantage of doing the testing in-house is having access to the test leaders, who can explain and clarify what has happened and the implications of it. This is obviously of primary interest to **D**.
- **Comprehensive documentation** – The documentation consists mainly of spreadsheets containing metrics and qualitative data. Metrics back up qualitative data and open up ways to present test results that can be understood without having to include contextual information. They make test re-

sults accessible for new groups. The quantitative data gives statistical confirmation of the early qualitative findings, but are regarded as most useful for PO, who want figures of the findings that have been reached. There is less pressure of time to get these results compiled, as the critical findings are already being implemented. The metrics can be subject to stringent analysis to show comparisons and correlations between different factors. In both organisations there is beginning to be a demand for Key Performance Indicators for usability, and although it is still unsure what these may consist of, it is still an indication of a trend that comes from **PO** level.

- **Customer collaboration** – in the testing procedure it is important for the testers to have easy access to individuals, to gain information about customer needs, end user patterns, etc. The whole idea of the test is to collect the information that is needed at the current time regarding the product and its development. How this is done in practice is obviously of concern to PO in the long run, but in the immediate day to day operation it is primarily of interest to **D**.
- **Contract negotiation** – On a high level it is up to PO to decide what sort of cooperation should take place between different organisations and customers, and this is not something that involves D, so this is seen as most important for **PO**.
- **Respond to change** – The test is easily adapted to changes, and is not particularly resource-intensive. If there is a need to change the format of a test, or a new test requirement turns up suddenly, it is easy to change the test without having expended extensive resources on the testing. It is also easy to do a "Light" version of a test to check a particular feature that arises in the everyday work of design, and this has happened several times at UIQ. This is the sort of thing that is a characteristic of the day to day work with interaction design, and is nothing that is of immediate concern for PO, so this is seen as **D**.
- **Following a plan** – From a short-term perspective, this is important for D, but since they work in a rapidly changing situation, it is more important for them to be able to respond to change. This is however important for **PO** who are responsible for well functioning strategies and long-term operations in the company.

5.1. On Opposite Sides of the Spectrum

In this analysis, we found that "Designers", as in the agile manifesto, are interested in the items on the left, rather than the items on the right (see Figure 2). We see this as being "A Designer's Manifesto". "Product Owners" are more interested in the items on the right. Boehm characterised the items on the right side as being "An Auditor Manifesto"[6]. We see it as being "A Product Owner's Manifesto". This is of course a sliding scale; some of the groups may be closer to the middle of the scale. Neither of the two groups is uninterested in what is happening at the opposite end of the spectrum, but as in the agile manifesto, while there is value in the items on one side, they value the items on the other side more. We are conscious of the fact that these two groups are very coarsely drawn, and that some groups and roles will lie between these extremes. We are unsure exactly which roles in the development process belong to which group, but are interested in looking at these extremes to see their information requirements in regard to the results of usability testing. On closer inspection it may be found that none of the groups is on the far side of the spectrum for all of the points in the manifesto. To gain further information regarding this, a case study has been performed, which we present in the next section.

6. Follow-up Study of Preferred Presentation Methods

This study is thus an investigation of attitudes regarding which types of usability findings different stakeholders need to see, and their pre-

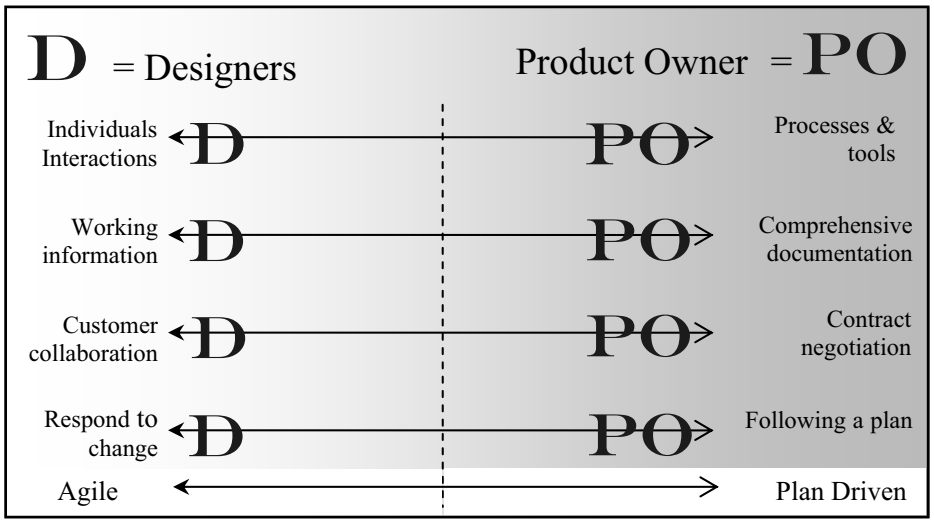


Figure 2. Groups and their diverging interests

ferred presentation methods. In the previous study we identified two groups of stakeholders with different information needs, ranging from Designers, who appear to want quick results, often qualitative results rather than quantitative results, to Product Owners, who want more detailed information, are more concerned with quantitative results, but are not as concerned with the speediness of the results. To test this theory, we sent a questionnaire to a number of stakeholders within UIQ and their customers, who are participants in the design and development process.

A document was compiled illustrating ten methods for presenting the results of UTUM tests. It contained a brief description of the presentation method and the information contained in it. The methods were chosen together with a usability expert from UIQ who often presents the results of testing to different groups of stakeholders. The methods were chosen on the basis of his experience of presenting test results to different stakeholders and are the most used and most representative ways of presenting results. The methods range from a verbal presentation of early findings, to spreadsheets containing all of the quantitative or the qualitative data from the testing, plus a number of graphical representations of the data. The methods were as follows

Method 1: The Structured Data Summary (the SDS). A spreadsheet with the qualitative findings of the testing. It shows issues that have been found, on the basis of each tester and each device, for every use case. Comments made by the test participants and observations made by the test leader are stored in the spreadsheet.

Method 2: A spreadsheet containing all "raw" data. All of the quantitative data from a series of tests. Worksheets contain the numerical data collected in a specific series of tests, which are also illustrated in a number of graphs. The data includes times taken to complete use cases, and the results of attitude assessments.

Method 3: A Curve diagram. A graph illustrating a comparison of time taken to complete one particular use case. One curve illustrates the average time for all tested telephones, and the other curves show the time taken for individual phones.

Method 4: Comparison of two factors (basic version). An image showing the results of a series of tests, where three telephones are rated and compared with regard to satisfaction and efficiency. No more information is given in this diagram.

Method 5: Comparison of two factors (brief details). The same image as Method 4, with a very brief explanation of the findings.

Method 6: Comparison of two factors (more in depth details). The same image as

Methods 4 and 5. Here, there is a more extensive explanation of the results, and the findings made by the test leader. The test leader has also written suggestions for short term and long term solutions to issues that have been found.

Method 7: The "Form Factor" – an immediate response. A visual comparison of which telephone was preferred by men and women, where the participants were asked to give an immediate response to the phones, and choose a favourite phone on the basis of "Form Factor" – the "pleasingness" of the design.

Method 8: PowerPoint presentation, no verbal presentation. A PowerPoint presentation, produced by the test leader. A summary of the main results is presented graphically and briefly in writing. This does not give the opportunity to ask follow-up questions in direct connection with the presentation.

Method 9: Verbal presentation supported by PowerPoint. A PowerPoint presentation, given by the test leader. A summary of the main results is presented graphically and briefly in writing, and explained verbally, giving the listener the chance to ask questions about e.g. the findings and suggestions for improvements. This type of presentation takes the longest to prepare and deliver.

Method 10: Verbal presentation of early results. The test leader gives a verbal presentation of the results of a series of tests. These are based mainly on his or her impressions of issues found, rather than an analysis of the metrics, and can be given after having observed a relatively small number of tests. This is the fastest and most informal type of presentation, and can be given early in the testing process.

The participants in the study were chosen together with the usability expert at UIQ. Some of the participants were people who are regularly given presentations of test results, whilst others were people who are not usually recipients of the results, but who in their professional roles could be assumed to have an interest in the results of usability testing. They were asked to read the document and complete the task by filling in their preferences in a spreadsheet. The results of the survey were returned to the researcher via e-mail, and have been summarised in a spreadsheet and then analysed on the basis of a number of criteria to see what general conclusions can be drawn from the answers.

The method whereby the participants were asked to prioritize the presentation methods was based on cumulative voting [19], [43], a well known voting system in the political and the corporate sphere ([13], [31]), also known as the $100 test or $100 method [20]. Cumulative voting is a method that has previously been used in the software engineering context, for e.g. software requirement prioritization [26] and the prioritization of process improvements [3], and in [2] where it is compared to and found to be superior to Analytical Hierarchy Process in several respects.

The questionnaire was sent to 29 people, mostly within UIQ but also to some people from UIQ's licensees. Only six respondents had replied to the questionnaire within the stipulated time, so one day after the first deadline, we sent out a reminder to the respondents who had not answered. This resulted in a further three replies. After one more week, we sent out a final reminder, leading to one more reply. Thus, we received 10 replies to the questionnaire, of which nine were from respondents within UIQ. On further enquiry, the reason given for not replying to the questionnaire was in general the fact that the company was in an intensive working phase for a planned product release, and that the staff at the company could not prioritise allocating the time needed to complete the questionnaire. This makes it impossible to give full answers to the research questions in this study, although it helps us to answer some of the questions, and gives us a better understanding of factors that affect the answers to the other questions. This study helps us formulate hypotheses for further work regarding these questions.

The division of roles amongst the respondents, and the number of respondents in the categories was as follows:

- 2: UI designers
- 2: Product planning
- 4: System design
- 1: Other (Usability)
- 1: Other (CTO Office)

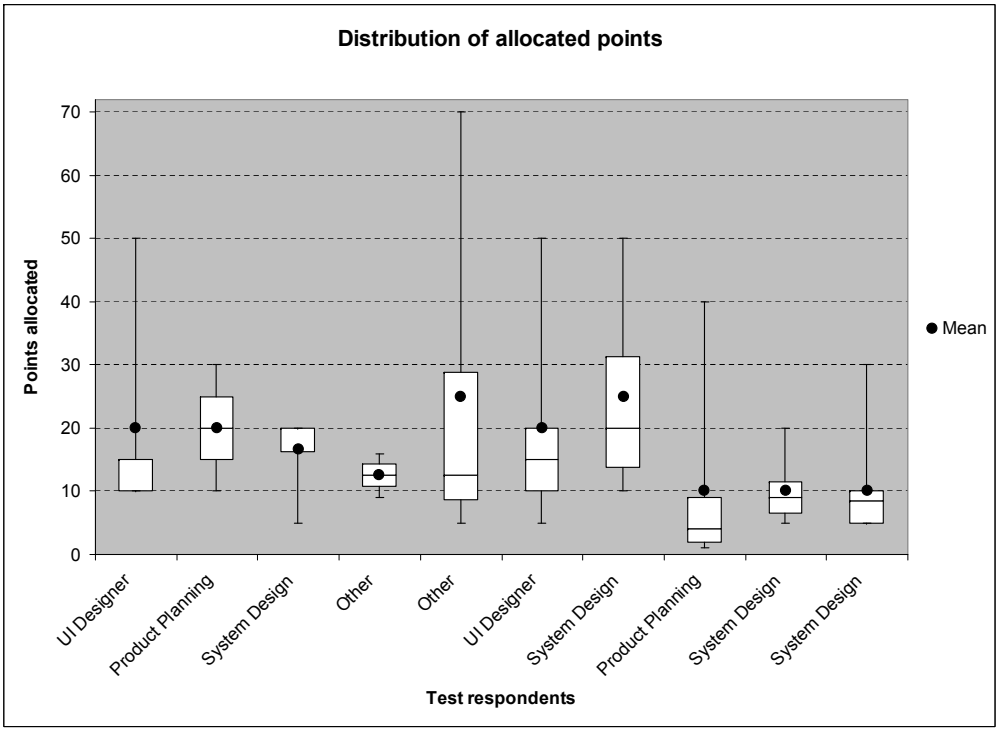


Figure 3. Distribution of points allocated per respondent

We have divided the respondents according to the tentative schema found in the first case study, between Designers (D) and Product Owners (PO). Some respondents were difficult to place in a particular category. The roles the respondents held in the company were discussed with a member of the management staff at UIQ, with long work experience at the company, who was well versed in the thoughts we had regarding the difference between Designers and Product Owners. Due to turbulence within the company, it was not possible to verify the respondents' attitudes to their positions, and would have been difficult, since they were not familiar with the terminology that we used, and the meaning of the roles that we had specified.

Five respondents, the two UI designers, the usability specialist and two of the system designers, belonged to the Designer group. The remaining five respondents, the two members of product planning, the respondent from the CTO office and two of the system designers, were representatives of the group of Product Owners.

Figure 3 is a box and whisker plot that shows the distribution of the points and the mean points allocated per person. As can be seen, the spread of points differs greatly from person to person. Although this reflects the actual needs of the respondent, the way of allocating points could also reflect tactical choices, or even the respondent's character. To get more information about how the choices were made would require a further study, where the respondents were interviewed concerning their strategies and choices.

In what follows, we use various ways of summarising the data. To obtain a composite picture of the respondents' attitudes, the methods are ranked according to a number of criteria. Given the small numbers of respondents in the study, this compilation of results is used to give a more complex picture of the results, rather than simply relying on one aspect of the questionnaire. The methods are ranked according to: the total number of points that were allocated by all respondents; the number of times the method has been chosen, and; the average ranking, which is the sum of the rankings given by each respondent, divided by the number of respondents that chose the method (e.g., if one respondent chose a method in first place, whilst another respondent chose it in third place, the average position is

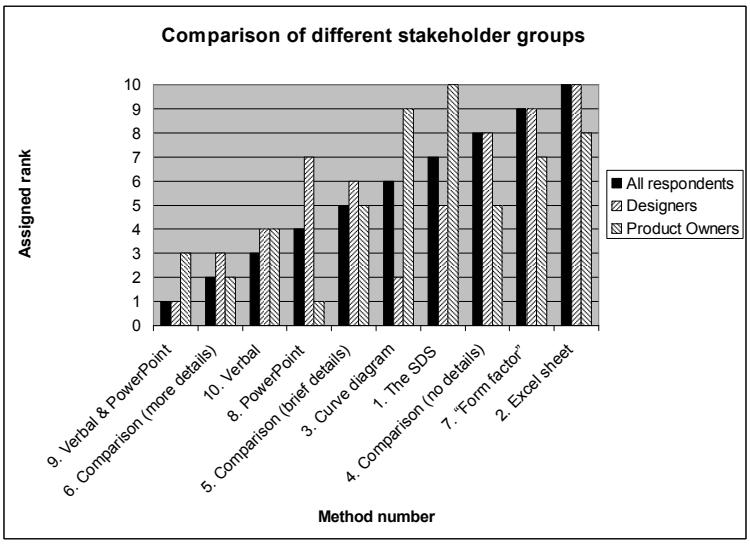


Figure 4. Comparison: All, Designers and Product Owners (lowest point is best)

$(1+3)/2 = 2$). A lower average ranking means a better result in the evaluation, although it gives no information about the number of times it has been chosen.

Figure 4 shows a summary of the results for all respondents and a comparison with the results for the group of Designers and Product Owners. For all respondents, total rank is very similar to ranking according to points allocated, and only two methods (ranked 5 and 6) have swapped places. Three methods head the list. Two are verbal presentations, one being supported by PowerPoint and the other is purely verbal.

Even within the two groups of Designers and Product Owners, there is little discrepancy between the results for total rank and position according to points awarded. Table 1 illustrates the ranks. The Methods are ordered according to the points allocated by all respondents. The next columns show the composite results, for all respondents and according to the two groups. Cases where the opinions differ significantly between Designers and Product Owners (a difference of 3 places or more) will be the subject of a brief discussion, to see whether we can draw any tentative conclusions about the presentation requirements of the different stakeholder groups. These methods, which are shown in italics in Table 1, are Methods 1, 3, 4 and 8. Since the company has now ceased operations, it is no longer possible to do a follow-up study of the attitudes of the participants, so the analysis is based on the knowledge we have of the operations at the company and the context where they worked. To verify these results, further studies are needed.

Method 3: The Curve Diagram. Designers ranked this presentation highly because if it is interpreted properly, it can give a great deal of information about the use case as it is performed on the device. If the device performs poorly in comparison to the other devices, which can easily be seen by the placement and shape of the curve, this indicates that there are problems that need to be investigated further. Use case performance time indicates the performance of the device, which can be correlated with user satisfaction. The shape of the curve illustrates when problems arose. If problems arise when performing the use case, these will be visible in the diagram and the Designers will know that there are issues that must be attended to.

Product Owners ranked this method poorly because the information is on the level of an individual use case, whilst they need information about the product or device at a coarser level of detail that is easy to interpret, giving an overall view of the product. They trust that problems at this level of detail are dealt with by the Designers, whilst they have responsibility for the product and process as a whole.

Table 1. Comparison of ranks: All, Designers and Product Owners

	Rankings			
Method	All respondents	Designers	Product Owners	Difference between groups
9. Verbal & PowerPoint	1	1	3	2
6. Comparison (more details)	2	3	2	1
10. Verbal	3	4	4	0
8. PowerPoint	4	7	1	6
5. Comparison (brief details)	6	6	5	1
3. Curve diagram	5	2	9	7
1. The SDS	7	5	10	5
4. Comparison (no details)	8	8	5	3
7. "Form factor"	9	9	7	2
2. Spreadsheet	10	10	8	2

Method 8: PowerPoint presentation, no verbal presentation. This can contain several ways of presenting the results of testing. Designers find this type of presentation of limited use because of the lack of contextual information and the lack of opportunity to pose follow-up questions. It gives a lot of information, but does not contain sufficient details to allow Designers to identify the problems or make decisions about solutions. Without details of the context and what happened in the testing situation, it is hard to interpret differences between devices, to know which problems there are in the device, and thereby difficult to know what to do about the problems. The length of time taken to produce the presentation also means that it is not suitable for Designers, who are concerned with fixing product issues as early in the development process as possible. We also believe that there is also a difference in "culture" where Designers are still unused to being presented with results in this fashion, and cannot translate this easily to fit in with their work practices.

This type of presentation is of primary interest to Product Owners because it provides an overall view of the product in comparison to other devices, without including too much information about the context and test situation. It contains sufficient text, and gives an indication of the status of the product. It is also adapted to viewing without the presence of the test leader, so the recipient can view the presentation and return to it at will. Product Owners are often schooled in an engineering tradition and are used to this way of presenting information.

Method 1: The Structured Data Summary (the SDS). Designers value this method of presentation because of the extent and character of the contextual information it includes, and because of the way the data is visualised. For every device and use case, there is information on issues that were observed, and records of comments made by the testers. It is easy to see which use cases were problematic, due to the number of comments written by the test leader, and the presence of many user comments also suggests that there are issues that need investigation. The contextual information gives clues to problems and issues that must be dealt with and gives hints on possible solutions. The effort required to read and summarise the information contained in the spreadsheet, leading to a degree of cognitive friction, means however that it is rated in the middle of the field rather than higher.

Product Owners rate this method poorly because they are uninterested in products on the level of use cases, which this presentation gives provides, and it is difficult to interpret for the device as a whole. The information is not adapted to the broad view of the product that the Product Owners need. The contextual information is difficult to summarise and does not give a readily understandable of the device as a whole. Product Owners find it difficult to make use of the information contained in this spread-

sheet and thereby rank it as least useful for their needs.

Method 4: Comparison of two factors (basic version). The lack of detail and of contextual information make if difficult for Designers to read any information that allows them to identify problems with the device. It simply provides them with a snapshot of how their product compares to other devices at a given moment.

Product Owners ranked this in the middle of the field. This is a simple way of visualising the state of the product at a given time, which is easy to compare over a period of time, to see whether a device is competitive with the other devices included in the comparison. This is typically one of the elements that are included in the PowerPoint presentation that Product Owners have ranked highest (Method 8). However, this particular method, when taken in isolation, lacks the richness of the overall picture given in Method 8 and is therefore ranked as lower.

To summarise these results, we find that the greatest difference between the two groups concerns the level of detail included in the presentation, the ease with which the information can be interpreted, and the presence of contextual information in the presentation. Designers prioritise methods that give specific information about the device and its features. Product Owners prioritise methods that give more overarching information about the product as a whole, and that is not dependent on including contextual information.

6.1. Changing Information Needs

Participants were informed that the survey was mainly focused on the presentation of results that are relevant during ongoing design and development. We pointed out that we believed that different presentation methods may be important in the starting and finishing phases of these processes. We stated that comments regarding this would be appreciated. Three respondents wrote comments about this factor.

One respondent (D) stated that the information needed in their everyday work as a UI designer, in the early stages of projects when the interaction designers are most active, was best satisfied through the verbal presentations of early results and verbal presentation supported by PowerPoint, whilst a non-verbal presentation, in conjunction with the metrics data in the spreadsheet and the SDS would be more appropriate later in the project, where the project activities were no longer as dependent on the work tasks and activities of the interaction designers.

A second respondent (D) stated that the verbal presentations are most appropriate in the requirements/design processes. Once the problem domain is understood, and the task is to iterate towards the best solution, the metrics data and the SDS would become more appropriate, because the problem is understood and the qualitative answers are more easily interpreted than the qualitative answers.

Another respondent (PO) wrote that it was important to move the focus from methods that were primarily concerned with verification towards methods that could be of assistance in requirements handling, in prioritisation and decision making in the early phases of development. In other words, the methods presented are most appropriate for later stages of a project, and there is a lack of appropriate methods for early stages.

Given the limited number of answers to these questions, it is of course difficult to draw any general conclusions, although it does appear to be the case that the verbal results are most important in the early stages of a project, to those who are involved in the actual work of designing and developing the product, whilst the more quantitative data is more useful as reference material in the later stages of a project, or further projects.

6.2. Attitudes Towards the Role of the Test Leader

The respondents were asked to judge whether or not they would need the help of the test leader in order to understand the presentation method in question. Two of the respondents supplied no answers to this question, and one of the respondents only supplied answers regarding

methods 9 and 10, which presuppose the presence of the test leader and are therefore excluded from the analysis. If we exclude these three respondents from the summary, there were seven respondents, of whom four gave answers for all eight methods, one gave five answers, and two gave three answers. The three respondents who did not answer these questions were all Product Owners, meaning that there were five designers and two Product Owners who answered these questions.

Analysis of the answers showed that, with the exception of Method 7 the methods that are primarily graphical representations of the data do not appear to require the presence of the test leader to explain the presentation. Method 7 was found to require the presence of the test leader, presumably because it was not directly concerned with the operations of the company. The spreadsheets however, one containing qualitative and one containing quantitative data, both require the presence of a test leader to explain the contents.

Given the fact that the Designers were in the majority, there were few obvious differences between Designers and Product Owners, although the most consistent findings here regard methods 4, 5, and 6, variations of the same presentation method with different amounts of written information. Here, Product Owners needed the test leader to be present whilst Designers did not.

6.2.1. In Summary

We now summarise the results of the research questions posed in this case study. The answer to the first question, whether any presentation methods are generally preferred, is that the respondents as a whole generally preferred verbal presentations. The primarily verbal methods are found in both first and third place. The most popular form was a PowerPoint presentation that was supported by verbal explanations of the findings. In second place is a non-verbal illustration showing a comparison of two factors, where detailed information is given explaining the diagram and the results it contains. This type of presentation is found in several variants in the study, and those with more explanatory detail are more popular than those with fewer details. Following these is a block of graphical presentation methods that are not designed to be dependent on verbal explanations. Amongst these is a spreadsheet containing qualitative data about the test results. At the bottom of the list is a spreadsheet that contains the quantitative data from the study. This presentation differs in character from the SDS, the spreadsheet containing qualitative data, since the SDS offers a view of the data that allows the identification of problem areas for the tested devices. This illustrates the fact that even a spreadsheet, if it offers a graphical illustration of the data that it contains, can also be found useful for stakeholders, even without an explicit explanation of the data that it contains.

Concerning the second question, we could identify differences between the two groups of stakeholders, and the greatest difference between the groups concerns the level of detail included in the presentation, the ease with which the information can be interpreted, and the presence of contextual information in the presentation. Designers prioritise methods that give specific information about the device and its features. Product Owners prioritise methods that give more overarching information about the product as a whole, and that is not dependent on including contextual information. We also found that both groups chose PowerPoint presentations as their preferred method, but that the Designers chose a presentation that was primarily verbal, whilst Product Owners preferred the purely visual presentation. Another aspect of this second question is the attitude towards the role of the test leader, where there were few obvious differences between Designers and Product Owners. The most consistent findings here concern variations of the same presentation method with different amounts of written information. Here, Product Owners needed the test leader to be present whilst Designers did not.

Regarding the third question, if there are methods that are lacking in the current presen-

tation methods, it was found that taking into account and visualising aspects of UX is becoming more important, and the results indicate that testing must be adapted to capture these aspects more implicitly. There is also a need for a composite presentation method combining the positive features of all of the current methods – however, given the fact that there do appear to be differences between information needs, it may be found to be difficult to devise one method that satisfies all groups.

No clear answers can be found for the fourth question, whether information needs, and preferred methods change during different phases of a design and development project. However, the replies suggest that the required methods do change during a project, that more verbally oriented and qualitative presentations are important in early stages of a project, in the concrete practice of design and development, and that quantitative orientated methods are important in later stages and as reference material.

Regarding the final question, whether results can be presented without the presence of the test leader, we find that the methods that are primarily graphical representations of the data do not appear to require the presence of the test leader to explain the presentation. The spreadsheets however, containing qualitative and quantitative data, both require the presence of a test leader to explain the contents.

To verify these results, further studies are of course needed. Despite the small scale of this study, the results give a basis for performing a further study, and allow us to formulate a hypothesis for following up our results. In line with the rest of the work performed as part of this research, we feel that the this work should be a survey based study in combination with an interview based study, in order to verify the results from the survey and gain a depth of information that is difficult to obtain from a purely survey based study.

We continue by discussing the results of the two case studies in relation to the industrial situation where we have been working, and the need for quality assurance in development and design processes.

7. Discussion

We begin by discussing our results in relation to academic discourses, to answer our first research question: *How can we balance demands for agile results with demands for formal results when performing usability testing for quality assurance?* We also comment upon two related discourses from the introductory chapter, i.e. the relation between quality and a need for cooperation between industry and research, and the relationship between quality and agility.

Since we work in a mass-market situation, and the system that we are looking at is too large and complex for a single customer to specify, the testing process must be flexible enough to accommodate the needs of many different stakeholders. The product must appeal to the broadest possible group, so it is difficult for customers to operate in dedicated mode with development team, with sufficient knowledge to span the whole range of the application, which is what an agile approach requires to work best [5]. In this case, test leaders work as proxies for the user in the mass market. We had a dedicated specialist test leader who brought in the knowledge that users have, in accordance with Pettichord [24]. Evidence suggests that drawing and learning from experience may be as important as taking a rational approach to testing [21]. The fact that the test leaders involved in the testing are usability experts working in the field in their everyday work activities means that they have considerable experience of their products and their field. They have specialist knowledge, gained over a period of time through interaction with end-users, customers, developers, and other parties that have an interest in the testing process and results. This is in line with the idea that agile methods get much of their agility from a reliance on tacit knowledge embodied in a team, rather than from knowledge written down in plans [5].

It would be difficult to gain acceptance of the test results within the whole organisation without the element of formalism. In sectors with large customer bases, companies require both rapid value and high assurance. This cannot be met by pure agility or plan-driven discipline; only a mix of these is sufficient, and organisations must evolve towards the mix that suits them best [5]. In our case this evolution has taken place during the whole period of the research cooperation, and has reached a phase where it has become apparent that this mix is desirable and even necessary.

In relation to the above, Osterweil [23] states that there is a body of knowledge that could do much to improve quality, but that there is "a yawning chasm separating practice from research that blocks needed improvements in both communities", thereby hindering quality. Practice is not as effective as it must be, and research suffers from a lack of validation of good ideas and redirection that result from serious use in the real world. This case study is part of a successful cooperation between research and industry, where the results enrich the work of both parts. Osterweil [23] also requests the identification of dimensions of quality and measures appropriate for it. The particular understanding of agility discussed in our case study can be an answer to this request. The agility of the test process is in accordance with the "good organisational reasons" for "bad testing" that are argued by Martin et al [21]. These authors state that testing research has concentrated mainly on improving the formal aspects of testing, such as measuring test coverage and designing tools to support testing. However, despite advances in formal and automated fault discovery and their adoption in industry, the principal approach for validation and verification appears to be demonstrating that the software is "good enough". Hence, improving formal aspects does not necessarily help to design the testing that most efficiently satisfies organisational needs and minimises the effort needed to perform testing. In the results of this work, the main reason for not adopting "best practice" is precisely to orient testing to meet organisational needs. Our case is a confirmation of [21]. Here, it is based on the dynamics of customer relationships, using limited effort in the most effective way, and the timing of software releases to the needs of customers as to which features to release. The present paper illustrates how this happens in industry, since the agile type of testing studied here is not according to "best practice" but is a complement that meets organisational needs for a mass-market product in a rapidly changing marketplace, with many different customers and end-users.

To summarise our second case study, the findings presented here are the results of a preliminary study that indicates the needs of different actors in the telecom industry. They are a validation of the ways in which UTUM results have been presented. They provide guidelines to improving the ways in which the results can be presented in the future. They are also a confirmation of the fact that there are different groups of stakeholders, the Designers and Product Owners found in our first case study, who have different information requirements. Further studies are obviously needed, but despite the small scale of this study, it is a basis for performing a wider and deeper study, and it lets us formulate a hypothesis regarding the presentation of testing results. We feel that the continuation of this work should be a survey based study in combination with an interview based study.

8. Conclusions and Further Work

In the usability evaluation framework, we have managed to implement a working balance between agility and plan driven formalism to satisfy practitioners in many roles. The industrial reality that has driven the development of this test package confirms the fact that quality and agility are vital for a company that is working in a rapidly changing environment, attempting to develop a product for a mass market. There is also an obvious need for formal data that can support the quick and agile results. The UTUM test package demonstrates one way to balance demands for agile results with demands for for-

mal results when performing usability testing for quality assurance. The test package conforms to both the Designer's manifesto, and the Product Owner's manifesto, and ensures that there is a mix of agility and formalism in the process.

The case in the present paper confirms the argumentation emphasizing 'good organizational reasons', since this type of testing is not according to "best practice" but is a complement that meets organisational needs for a mass-market product in a rapidly changing marketplace, with many different customers and end-users. This is partly an illustration of the chasm between industry and research, and partly an illustration of how agile approaches are taken to adjust to industrial reality. In relation to the former this case study is a successful cooperation between research and industry. It has been ongoing since 2001, and the work has an impact in industry, and results enrich the work of both parts. The inclusion of Sony Ericsson in this case study gave even greater possibilities to spread the benefits of the cooperative research. More and more hybrid methods are emerging, where agile and plan driven methods are combined, and success stories are beginning to emerge. We see the results of this case study and the UTUM test as being one of these success stories. How do we know that the test is successful? By seeing that it is in successful use in everyday practice in an industrial environment. We have found a successful balance between agility and formalism that works in industry and that exhibits qualities that can be of interest to both the agile and the software engineering community.

Acknowledgements This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the software development project "Blekinge – Engineering Software Qualities", www.bth.se/besq. Thanks to the participants in the study and to my colleagues in the U ODD research group for their help in data analysis and structuring my writing. Thanks also to Gary Denman for permission to use the Structured Data Summary.

References

[1] K. Beck. *Extreme Programming Explained.* Addison Wesley, Reading, MA, 2000.

[2] P. Berander and P. Jönsson. Hierarchical cumulative voting (HCV) – prioritization of requirements in hierarchies. *International Journal of Software Engineering & Knowledge Engineering*, 16(6):819, 2006.

[3] P. Berander and C. Wohlin. Identification of key factors in software process management – a case study. In *2003 International Symposium on Empirical Software Engineering, ISESE '03*, pages 316–325, Rome, Italy, 2003.

[4] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[5] B. W. Boehm. Get ready for agile methods, with care. *Computer*, 35(1):64–69, 2002.

[6] B. W. Boehm. Keynote address, 5th Workshop on Software Quality (WoSQ), 2007.

[7] J. Brooke. System usability scale (SUS): a Quick-and-Dirty method of system evaluation user information, 1986.

[8] BTH. UIQ, usability test. http://www.youtube.com/watch?v=5IjIRlVwgeo, Aug. 2008.

[9] A. Cockburn and J. Highsmith. *Agile Software Development.* The Agile Software Development Series. Addison-Wesley, Boston, 2002.

[10] Y. Dittrich, C. Floyd, and R. Klischewski. *Doing Empirical Research in Software Engineering: finding a path between understanding, intervention and method development*, pages 243–262. MIT Press, 2002.

[11] Y. Dittrich, K. Rönkkö, J. Erickson, C. Hansson, and O. Lindeberg. Co-operative method development: Combining qualitative empirical research with method, technique and process improvement. *Journal of Empirical Software Engineering*, 2007.

[12] Y. Dittrich, K. Rönkkö, O. Lindeberg, J. Erickson, and C. Hansson. Co-operative method development revisited. *SIGSOFT Softw. Eng. Notes*, 30(4):1–3, 2005.

[13] J. N. Gordon. Institutions as relational investors: A new look at cumulative voting. *Columbia Law Review*, 94(4):124–193, 1994.

[14] M. J. Harrold. Testing: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, Limmerick, Ireland, 2000. ACM Press.

[15] M. Hassenzahl, E. L. Law, and E. T. Hvannberg. User experience – towards a unified view. In

UX WS NordiCHI'06, pages 1–3, Oslo, Norway, 2006. cost294.org.

[16] M. Hassenzahl and N. Tractinsky. User experience – a research agenda. *Behaviour & Information Technology*, 25(2):91–97, 2006.

[17] International Organization for Standardization. ISO 9241-11 (1998): Ergonomic requirements for office work with visual display terminals (VDTs) – part 11: Guidance on usability. Technical report, 1998.

[18] International Organization for Standardization. ISO 9126-1 software engineering – product quality – part 1: Quality model, 2001.

[19] Investopedia.com. Cumulative voting. http://www.investopedia.com/terms/c/cumulativevoting.asp, Apr. 2009.

[20] D. Leffingwell and D. Widrig. *Managing Software Requirements: A Use Case Approach*, volume 2nd. Addison Wesley, 2003.

[21] D. Martin, J. Rooksby, M. Rouncefield, and I. Sommerville. 'Good' organisational reasons for 'Bad' software testing: An ethnographic study of testing in a small software company. In *ICSE '07*, Minneapolis, MN, 2007. IEEE.

[22] E. Mumford. Advice for an action researcher. *Information Technology and People*, 14(1):12–27, 2001.

[23] L. Osterweil. Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, 28(4):738–750, 1996.

[24] B. Pettichord. Testers and developers think differently. *STGE magazine*, Vol. 2(Jan/Feb 2000 (Issue 1)), 2000.

[25] S. L. Pfleeger and J. M. Atlee. *Software Engineering*, volume 3rd. Prentice Hall, Upper Saddle River, NJ, 2006.

[26] B. Regnell, M. Höst, J. N. och Dag, P. Beremark, and T. Hjelm. An industrial case study on distributed prioritisation in Market-Driven requirements engineering for packaged software. *Requirements Engineering*, 6(1):51–62, 2001.

[27] C. Robson. *Real World Research*, volume 2nd. Blackwell Publishing, Oxford, 1993.

[28] K. Rönkkö. *Making Methods Work in Software Engineering: Method Deployment as a Social achievement*. PhD thesis, Blekinge Institute of Technology, School of Engineering, 2005. Dissertation Series No. 2005:04; Doctoral Thesis.

[29] K. Rönkkö. Ethnography. In P. Laplante, editor, *Encyclopedia of Software Engineering*. Taylor and Francis Group, New York, 2008.

[30] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *9th international conference on Software Engineering*, pages 328–338, Monterey, California, United States, 1987. IEEE Computer Society Press.

[31] J. Sawyer and D. McRae, Jr. Game theory and cumulative voting in Illinois: 1902–1954. *The American Political Science Review*, 56(4):936–946, 1994.

[32] D. Schuler and A. Namioka. *Participatory Design – Principles and Practices*, volume 1st. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1993.

[33] I. Sommerville. *Software Engineering*, volume 8. Addison Wesley, 1982.

[34] D. Talby, O. Hazzan, Y. Dubinsky, and A. Keren. Agile software testing in a Large-Scale project. *IEEE Software*, 23(4):30–37, 2006.

[35] The Agile Alliance. The agile manifesto. http://agilemanifesto.org/, Apr. 2009.

[36] The Agile Alliance. Principles of agile software. http://www.agilemanifesto.org/principles.html, Apr. 2009.

[37] U-ODD. Use-Oriented Design and Development. http://www.bth.se/tek/u-odd, Apr. 2009.

[38] UIQ Technology. Company information. http://uiq.com/aboutus.html, June 2008.

[39] UIQ Technology. UIQ Technology Usability Metrics. http://uiq.com/utum.html, June 2008.

[40] UIQ Technology. UTUM website. http://uiq.com/utum.html, June 2008.

[41] UXEM. User eXperience Evaluation Methods in product development (UXEM). http://www.cs.tut.fi/ihte/CHI08_workshop/slides/Poster_UXEM_CHI08_V1.1.pdf, June 2008.

[42] UXNet: the user experience network. http://uxnet.org/, June 2008.

[43] Wikipedia. Cumulative voting. http://en.wikipedia.org/wiki/Cumulative_voting, Apr. 2009.

[44] J. Winter, K. Rönkkö, M. Ahlberg, M. Hinely, and M. Hellman. Developing quality through measuring usability: The UTUM test package. In *ICSE 2007*, 5th Workshop on Software Quality, at ICSE 2007, 2007.

[45] WoSQ. Fifth workshop on software quality, at ICSE 07. http://attend.it.uts.edu.au/icse2007/, June 2008.

[46] R. K. Yin and S. Robinson. *Case Study Research – Design and Methods*, volume 3rd of *Applied Social Research Methods Series*. SAGE publications, 5, 2003.

e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Web–Server Systems HTCPNs-Based Development Tool Application in Load Balance Modelling

Slawomir Samolej*, Tomasz Szmuc**
*Department of Computer and Control Engineering, Rzeszów University of Technology
**Institute of Automatics, AGH University of Science and Technology
ssamolej@prz.edu.pl, tsz@agh.edu.pl

Abstract

A new software tool for web–server systems development is presented. The tool consist of a set of predefined Hierarchical Timed Coloured Petri Net (HTCPN) structures – patterns. The patterns make it possible to naturally construct typical and experimental server–systems structures. The preliminary patterns are executable queueing systems. A simulation based methodology of web–server model analysis and validation has been proposed. The paper focuses on presenting the construction of the software tool and its application for selected cluster–based web–servers load balancing strategies evaluation.

1. Introduction

Gradually, the Internet becomes the most important medium for conducting business, selling services and remote control of industrial processes. Typical modern software applications have a client–server logical structure where predominant role plays an Internet server offering data access or computation abilities for remote clients. The hardware of an Internet or web–server is now usually designed as a set of (locally) deployed computers. The computers are divided into some layers or clusters where each layer executes separate web–system task [4], [12], [24], [34], [39], [37], [5], [2], [22], [9], [29], [23]. This design approach makes it possible to distribute services among the nodes of a cluster and to improve the scalability of the system. Redundancy which intrinsically exists in such hardware structure provides higher system dependability. Fig. 1 shows an example cluster–based Internet system structure. The Internet requests are generated by the clients. Then they are distributed by the load balancer among set of computers that constitute the front-end or WWW cluster. The front–end cluster offers a system interface and some procedures that optimize the load of the next system layer–the database server.

To improve the quality of service of web–server clusters two main research paths are followed. First, the software of individual web–server nodes is modified to offer average response time to dedicated classes of consumers [11], [18], [19]. Second, some distribution strategies of cluster nodes are investigated [4], [29] in conjunction with searching for load balancing policies for the nodes [6], [32], [39], [37], [5], [2], [22]. In several research projects reported in [12], [30], [34] load balancing algorithms and modified cluster node structures are analyzed together.

It is worth noticing that in some of abovementioned manuscripts searching for a solution of the problem goes together with searching for the adequate formal language to express the system developed [3], [12], [30], [32], [34], [39]. In [3], [32], [34], [39] Queueing Nets whereas in [30] Stochastic Petri Nets are applied for

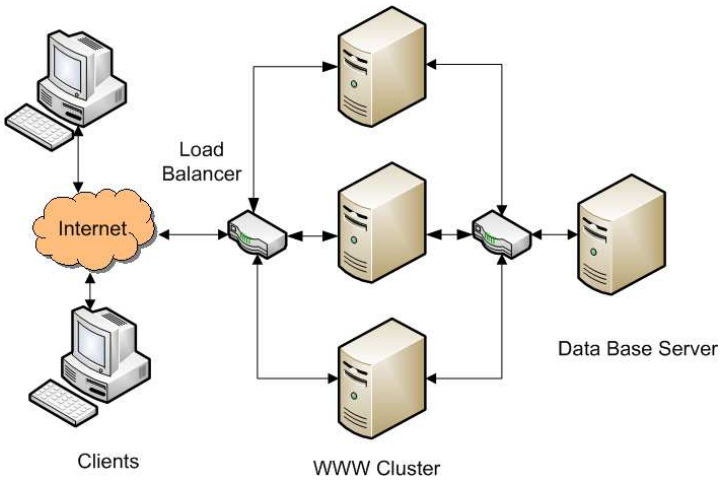


Figure 1. Example distributed cluster–based Internet system

system model construction and examination. However, the most mature and expressive language proposed for the web–cluster modelling seems to be Queueing Petri Nets (QPNs) [12]. The nets combine coloured and stochastic Petri nets with queueing systems [1] and consequently make it possible to model relatively complex web–server systems in a concise way. Moreover, there exists a software tool for the nets simulation [13]. The research results reported in [12] include a systematic approach to applying QPNs in distributed applications modelling and evaluation. The modelling process has been divided into following stages: system components and resources modelling, workload modelling, intercomponent interactions and processing steps modelling, and finally – model parameterization. The final QPNs based model can be executed and used for modelled system performance prediction.

The successful application of QPNs in web–cluster modelling become motivation to research reported in this paper. The aim of the research is to provide an alternative methodology and software tool for cluster–based hardware/software systems development. The main features of the methodology are as follows:

– The modelling language will be Hierarchical Timed Coloured Petri Nets (HTCPNs) [7],
– A set of so called HTCPNs design patterns (predefined net structures) will be prepared and validated to model typical web cluster components,
– The basic patterns will be executable models of queueing systems,
– A set of design rules will be provided to cope with the patterns during the system model creation,
– The final model will be an executable and analyzable Hierarchical Timed Coloured Petri Net,
– A well established Design/CPN and CPN Tools software toolkits will be used for the design patterns construction and validation,
– The toolkits will also be used as a platform for the web–server modelling and development,
– Performance analysis modules of the toolkits will be used for capturing and monitoring the state of the net during execution.

The choice of HTCPNs formalism as a modelling language comes from the following prerequisites. First, HTCPNs have an expression power comparable to QPNs. Second, the available software toolkits for HTCPNs composition and validation seem to be more popular than“SimQPN” [13]. Third, there exists a reach knowledge base of successful HTCPNs applications to modelling and validation of wide range software/hardware systems [7] including web–servers [24], [27], [36]. The rest named features of design methodology introduced in this paper results from both generally known capabilities of software toolkits for HTCPNs modelling and some previous experience gained by the authors in applica-

tion HTCPNs to real–time systems development [25], [26].

This paper is organized as follows. Section 2 describes some selected design patterns and rules of applying them to web–server cluster model construction. An example queueing system, web–server subsystem and top–level system models are presented. Then the simulation based HTCPNs models validation methods are discussed. Section 3 presents HTCPNs models of selected experimental and applied load balancing strategies for computer clusters. The load balancing models construction and some simulation results are discussed. Conclusions and future research program complete the paper.

It has been assumed that the reader is familiar with the basic principles of Hierarchical Timed Coloured Petri Nets theory [7], [8], [14]. All the Coloured Petri Nets in the paper have been edited and analysed using Design/CPN tool [21], [36]. Equivalent HTCPNs models may be developed using CPN Tools [8], [35] software toolkit.

2. Cluster Server Modelling Methodology

The main concept of the methodology lies in the definition of reusable timed coloured Petri nets structures (patterns) making it possible to compose web–server models in a systematic manner. The basic set of the patterns includes typical *queueing systems* TCPNs implementations, eg. –/M/PS/∞ , –/M/FIFO/∞ [24], [27]. *Packet distribution* TCPNs *patterns* constitute the next group of reusable blocks. They preliminary role is to provide some predefined web–server cluster substructures composed from the queueing systems. At this stage of subsystem modelling the queueing systems are represented as substitution transitions (compare [24], [27]). The separate models of system arrival processes are also the members of the group mentioned. The *packet distribution patterns* represented as substitution transitions are in turn used for the general *top–level system model* composition. As a result, the 3–level web–server model composition has been proposed. The top–level TCPN represents the general view of system components. The middle–level TCPNs structures represent the queueing systems interconnections. And the lowest level includes executable queueing systems implementations.

The modelling methodology assumes, that the actual state of the Internet requests servicing in the system can be monitored. Moreover, from the logical point of view the model of the server cluster is an open queueing network, so the requests are generated, serviced and finally removed from the system. As a result an important component of the software tool for server cluster development is *the logical representation of the requests*.

In the next subsections the following features of the modelling methodology will be explained in detail. First, the logical representation of Internet requests will be shown. Second, queueing system modelling rules will be explained. Third, an example cluster subsystem with an individual load–balancing strategy will be proposed. Fourth, Internet request generator structure will be examined. Fifth, top–level HTCPNs structure of an example cluster–server model will be shown. Finally, model analysis capabilities will be discussed.

2.1. Logical Request Representation

In the server–cluster modelling methodology that is introducing in the paper the structure of the HTCPN represents a hardvare/software architecture of web–server. Yet, the dynamics of the modelled system behavior is determined by state and allocation of tokens in the net structure. Two groups of tokens has been proposed for model construction. The first group consists of the so–called local tokens, that "live" in individual design patters. They provide local functions and data structures for the patterns. The second group of tokens represents Internet requests that are serviced in the system. They are transported throughout several cluster components. Their internal state carries data that may be used for timing and performance evaluation of the system modelled. As the tokens represent-

ing the requests have the predominant role in the modelling methodology, they structure will be explained in detail.

Each token representing an Internet request is a tuple

$$PACKAGE = (ID, PRT, START\_TIME, PROB, AUTIL, RUTIL),$$

where *ID* is a *request identifier*, *PRT* is a *request priority*, *START_TIME* is a *value of simulation time when the request is generated*, *PROB* is a *random value*, *AUTIL* is an *absolute request utilization value*, and *RUTIL* is a *relative request utilization value*. *Request identifier* makes it possible to give the request an unique number. *Request priority* is an integer value that may be taken into consideration when the requests are scheduled according priority driven strategy [11]. *START_TIME* parameter can store a simulation time value and can be used for the timing validation of the requests. *Absolute request utilization value*, and *relative request utilization value* are exploited in some queueing systems execution models (e.g. with processor sharing service).

2.2. Queueing System Models

The basic components of the software tool for web–server clusters development introduced in this paper are the executable queueing systems models. At the current state of the software tool construction the queueing systems models can have *FIFO*, *LIFO*, *processor sharing* or *priority based* service discipline. For each queue an arbitrary number of service units may be defined. Additionally, the basic queueing systems has been equipped with auxiliary components that are responsible for monitoring of internal states of the queue during its execution.

An example HTCPNs based –/1/FIFO/∞ queueing system model is shown in Fig. 2. The model is a HTCPNs subpage that can communicate with the parent page via *INPUT_PACKS*, *OUTPUT_PACKS* and *QL* port places. Request packets (that arrive through *INPUT_PACK* place) are placed into a queue structure within *PACK_QUEUE* place after *ADD_FIFO* transition execution. *TIMERS* place and *REMOVE_FIFO* transition constitute a clock–like structure and are used for modelling of duration of packet execution. When *REMOVE_FIFO* transition fires, then the first packet from the queue is withdrawn and directed to the service procedure.

The packets under service acquire the adequate time stamps generated according to the assumed service time random distribution function. The time stamps associated with the tokens prevent from using the packet tuples (the tokens) for any transition firing until the stated simulation time elapses (according to firing rules defined for HTCPNs [7]). The packets are treated as serviced when they can leave *OUTPUT_PACKS* place as their time stamps expired. The number of tokens in TIMERS place defines the quantity of queue servicing units in the system.

Main parameters that define the queueing system model dynamics are queue mean service time, service time probability distribution function and number of servicing units. Capacity of the queue is not now taken into consideration and theoretically may be unlimited.

For future applications the primary queueing system design pattern explained above has been equipped with an auxiliary "plug–in". *COUNT_QL* transition and *TIMER_QL*, *QL* and *COUNTER* places make it possible to measure the queue length and export the measured value to the parent CPNs page during the net execution. *TIMER_QL* place includes a timer token that can periodically enable the *COUNT_QL* transition. *QL* port place includes a token storing the last measured queue length and an individual number of a queueing system in the system. The *COUNTER* place includes a counter token used for the synchronization purpose.

2.3. Packet Distribution Models

Having a set of *queueing systems design patterns* some *packet distribution HTCPNs structures* may be proposed. In [24] a typical homogeneous multi–tier web–server structure pat-

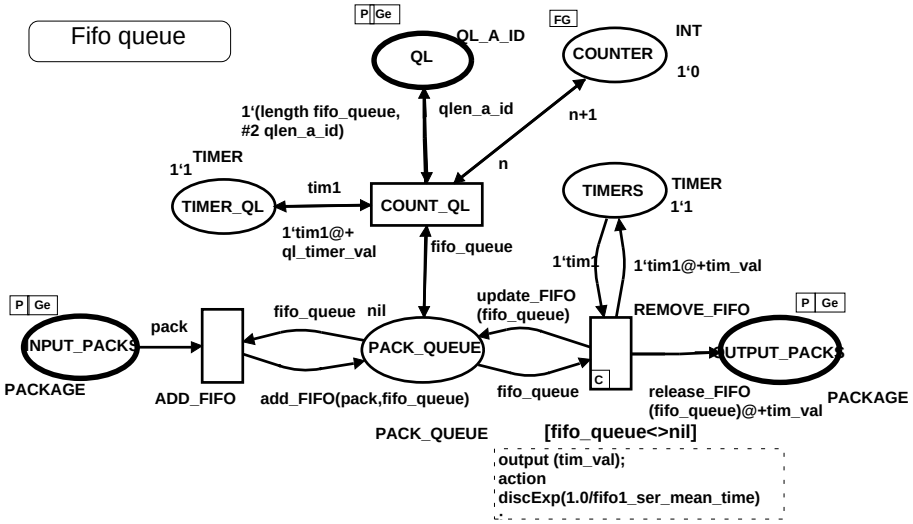


Figure 2. HTCPNs based –/1/FIFO/∞ queueing system model

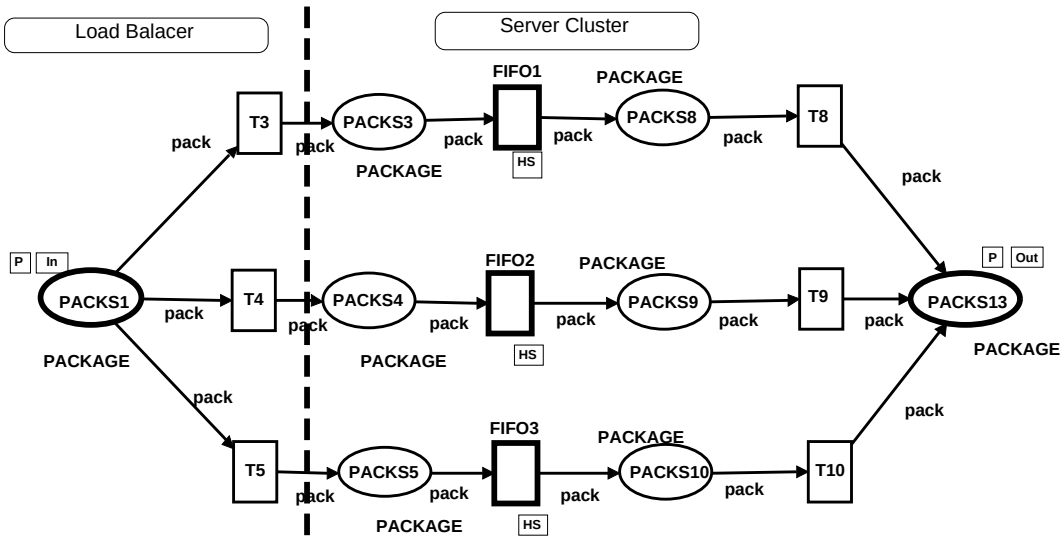


Figure 3. WWW cluster with stochastic load balancer

tern was examined, in [23] a detailed distributed database cluster model was proposed, whereas in [27] a preliminary version of server structure with feedback like admission control of Internet requests was introduced. The packet distribution patterns presented in this paper are also related to the load balancing in web–server cluster problem. The detailed discussion of some selected load balancing strategies models is included in section 3. In this section a simple WWW cluster model with stochastic packed distribution policy is concerned.

Figure 3 includes an example of cluster load–balancing HTCPNs model. The cluster consists of 3 computers (compare Fig. 1) represented as *FIFO1*...*FIFO3* substitution transitions, where each transition is attached to the corresponding *FIFO* queueing pattern. The Internet requests serviced by the cluster arrive through *PACKS1* port place. A *load balancer* decides where the currently acquired request should be send. When a token arrives in *PACKS1* place, transitions *T3*...*T5* are in conflict. According to CPN properties, a transition is randomly chosen for firing. Consequently, the stochastic packet distribution policy is naturally modelled.

2.4. Request Generator Model

According to one of main assumptions of the web–server cluster modelling methodology presented in this paper, the system model can be treated as an open queueing network. Consequently, the crucial model component must be a network arrival process simulating the Internet service requests that are sent to the server.

Figure 4 shows an example HTCPNs subpage that models a typical Internet request generator. The core of the packet generator is a clock composed from *TIMER0* place and *T0* transition. The code segment attached to the *T0* transition produces values of time–stamps for tokens stored in *TIMER0* place. The values are defined by the defined probability function. As a result the Internet requests appear into *PACKS1* place at random moments in simulation time. The frequency at which tokens appear in *PACKS1* place is determined by the mentioned above distribution function. *PACKS1* place has a port place status and thereafter tokens appearing in it can be consumed by other model components (e.g. server cluster model).

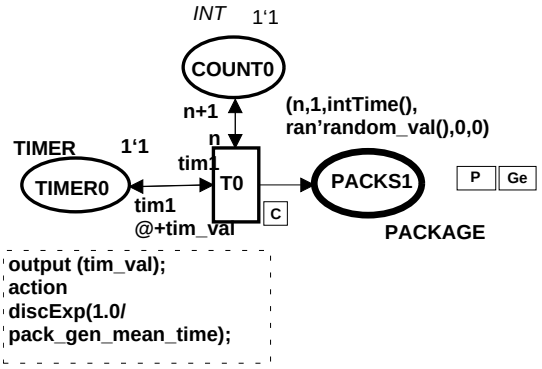


Figure 4. Web–server arrival process model

The Internet request frequency can have any standard probability distribution function or can be individually constructed as it was proposed in [36].

2.5. Example Top–Level Multi–Tier Server Model

Having the adequate set of design patterns, a wide area of server cluster architectures can be modelled and tested at the early stage of development process. At the top–level modelling process each of the main components of the system can be represented as a HTCPNs substitution transition. The modelling methodology presented in the paper suggest that at the top–level model construction the arrival process and main server cluster layers should be highlighted. After that each of the main components (main substitution transition) should be decomposed into an adequate packed distribution subpage, were under some of transitions queueing system models will be attached. It is easily to notice that a typical top–down modelling approach of software/hardware system modelling has been adapted in the web server modelling methodology proposed in the paper.

Figure 5 includes an example top–level HTCPN model of cluster–based server (compare Fig. 1) that follows the abovementioned modelling development rules. The HTCPN in Fig. 5 consists of 3 substitution transitions. *Input_Procs* transition represents the arrival process for the server cluster, whereas *WWW_Server_Cluster* transition represents the first–layer of multi–tier web–server, and finally *DataBaseServer* transition represents the data base server.

The modelling process can be easily extended by attaching the request generator model as in section 2.4 under the *Input_Procs* transition and by attaching the WWW cluster model with load balancing module as in section 2.3 under *WWW_Server_Cluster* transition. The final executable model can be acquired by attaching FIFO design patterns under $FIFO1$, $FIFO2$ and $FIFO3$ transitions in the load balancing module (compare sections 2.2 and 2.3). A separate model should be proposed for the packet distribution and queueing models layers of the data base server.

2.6. Model Validation Capabilities

Typical elements of HTCPNs modelling software tools are performance evaluation routines, e.g.: [16], [35] . The routines make it possible to capture the state of dedicated tokens or places

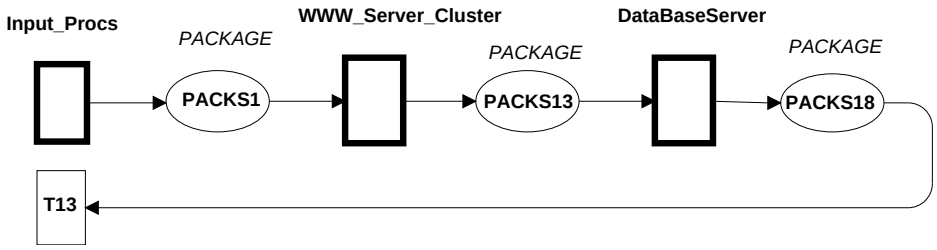


Figure 5. Example top–level multi–tier server model

during the HTCPN execution. A special kind of log files showing the changes in the state of HTCPN can be received and analyzed offline.

At the currently reported version of web–server cluster modelling and analysis software tool, queue lengths and service time lengths can be stored during the model execution. Detecting the queue lengths seems to be the most natural load measure available in typical software systems. The service time lengths are measurable in the proposed modelling method because of a special kind *PACKAGE* type tokens construction (compare section 2.1). The tokens "remember" the simulation time at which their appear in the cluster and thereafter the time at each state of their service may be captured. In real systems the service time is a predominant quality of service parameter for performance evaluation.

The performance analysis of models of web servers constructed according the proposed in the paper methodology can be applied in the following domains.

First, the system instability may be easily detected. The stable or balanced queueing system in a steady state has an approximately constans average queue length and correspondingly average service time. On the contrary, when the arrival process is to intensive for the queueing systems to serve, both queue lengths and service times increase. This kind of analysis is possible when there are no limitations for queue lengths in the proposed modelling method. Fig. 6 shows the queue lengths (Fig. 6 (left)) and service time lengths (Fig. 6 (right)) when the considered web server cluster model is permanently overloaded.

Second, the average values of queueing system systems parameters such as average queue lengths and average servicing times for the balanced model can be estimated. Provided that the arrival process model and the server nodes models parameters are acquired from the real devices as in [18], [30], [34], [36], the software model can be used for derivation the system properties under different load conditions. In the Fig. 7 queue lengths (Fig. 7 (left)) and service times (Fig. 7 (right)) under stable system execution are shown. The cluster had a heterogeneous structure, where server 2 (FIFO2 model) had 4 times lower performance. FIFO1 and FIFO3 average queue length was 1.7, whereas FIFO3 queue length was 4.4. The average service time for FIFO1 and FIFO3 cluster nodes was 811 time units whereas for FIFO2 was 7471 time units.

Third, some individual properties of cluster node structures or load balancing strategies may be observed. Some selected load balancing algorithms properties derived from simulation experiments will be discussed in section 3.

3. Example of Load Balancing Strategies Evaluation

Load balancing is an important issue in parallel and distributed systems. In traditional computation systems load balancing procedures were used to distribute the computation task among system nodes. It improved the general system utilisation and usually led to faster processing. In recent years, the load balancing algorithms elaborated for general parallel and distributed systems [31] were naturally re-applied in the emerging locally distributed Internet or web systems. The first load balancing strategies successively applied in Internet systems were static random distribution policy [28] and static modulus-based round-robin

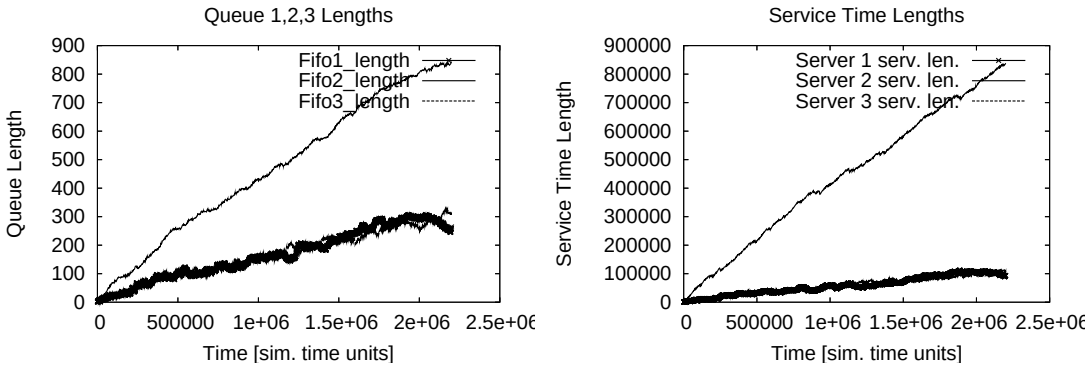


Figure 6. Queue lengths (left) and service times (right) under overload condition

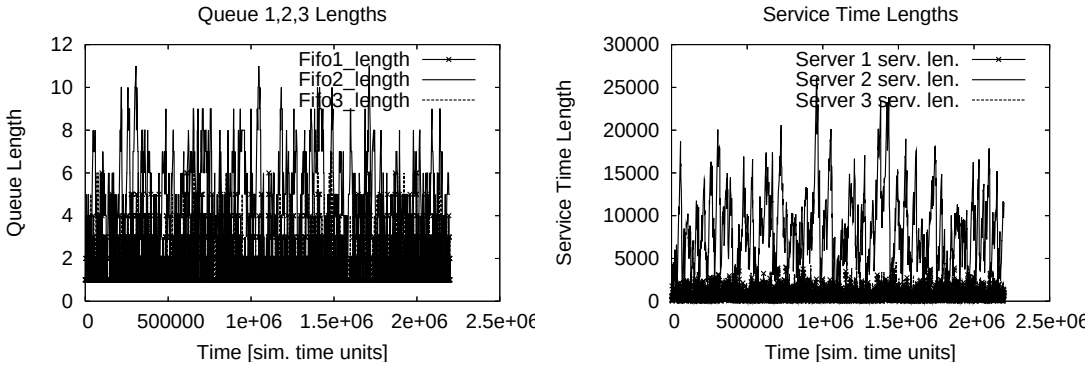


Figure 7. Queue lengths (left) and service times (right) under stable system execution

policy [15]. There were described in the paper [38] successful implementations of round-robin, weighted round-robin, least-connection and weight least-connection load balancing strategies in Linux Virtual Server [17] . The abovementioned strategies are now standard algorithms in commercial load balancing solutions as mentioned in [20], [33].

The rapid development of web–server oriented load balancing policies was tentatively systematised in [5], [2]. Paper [5] classifies distributed web-server architectures regarding the entity which distributes requests among servers. It defines 4 approaches of request distribution: client–based, DNS–based, dispatcher–based, and server–based. In [2] an attempt of simulation based on comparison of selected load balancing algorithms such as "round robin", "least connection first", "round-trip" and "Xmitbyte" was carried out.

During last few years research has focused on the so-called dynamic load balancing strategies for web–oriented systems. Generally, the dynamic load balancing polices use some kind of feedback information from the cluster nodes to redirect the incoming request among the nodes. In [30] the so called "fewest server processes first" and "extended fewest server processes first" dynamic load balancing policies were compared. The fewest server processes first policy concept is (in our opinion) comparable to least–connection policy. In both algorithms the least loaded server gets the next incoming request. The "extension" of the preliminary algorithm lies in the fact that the request can have priorities. The paper presents high-level Petri net approach to efficiency analysis of the policies in different priority levels scenarios. A dynamic web system load balancing policy presented in [37] (AdaptLoad policy) adopts the load of the servers according to size of documents requested. The policy builds a discrete data histogram encoding empirical size distribution of batches of K requests as they ar-

rive in the system. Each server "offers" files within a certain range of size. The range depends of "popularity" of the files derived from the histogram. The paper presents simulation results of the policy behaviour under historical load conditions. Paper [6] at first discusses such so called nonprediction–based load balancing techiques as "first fit", "stream–based mapping" and "adaptive load sharing" correspondingly. The techniques are examined with respect to possible application in multimedia applications. The prediction–based load balancing techniques as "least load first", "prediction–based least load first", "adaptive partition" and "prediction–based adaptive partition" are introduced and experimentally evaluated. In [22] a sum of weighted factors such as CPU usage, memory usage, number of processors, number of I/O operations, amount of free local storage and network I/O usage are taken into consideration to compute the load of a cluster node. The load of the node may then be applied to a load balancing policy.

At the current state of the development of the HTCPNs–based tool, some selected load balancing HTCPNs templates were modelled. Three of the most widely applied polices such as "random", "round-robin" and "fewest server processes first" were implemented. Additionally one experimental–"adaptive load sharing" policy was chosen for the implementation, because as it was claimed in [6], this policy offers reasonable balance between the throughput and out–of–order departures of the external requests. In the following subsections the HTCPNs–based models of the mentioned load balancing policies will be presented. The final subsection will include some simulation results of the HTCPNs–based load balancing algorithms. The model of the simplest– "random" load balancing policy was presented in subsection 2.3.

3.1. Round-robin Load Balancing Policy Model

Figure 8 presents HTCPNs–based model of the computer cluster similar to the cluster model in Fig. 3. The cluster consists of 3 computers servicing requests incoming via $PACKS1$ port place. The incoming Internet requests are redistributed among the cluster nodes according to round-robin load balancing policy. The model of the policy works as follows. Each incoming packet "passes" $T2$ transition and after the transition firing the forth element of the tuple modelling the requests (see subsection 2.1) is modified. The element includes a number of the server where the packet will be serviced. Guard functions attached to $T3$, $T4$, $T5$ transitions "check" the fourth element of each packed model and "pass" the related requests. The presented load balancing policy model can be easily extended to "weighted round-robin" policy by extending the numbers generated for the forth element of the packet tuple and by the corresponding modifications of the guards.

3.2. Fewest Server Processes First Load Balancing Policy Model

Figure 9 includes HTCPNs–based model of the computer cluster similar to the cluster model in Fig. 3 and Fig. 8. The incoming Internet requests are redistributed among the cluster nodes according fewest serwer processes first load balancing policy. The the model of the policy works as follows. During the model execution, the lengths of the queues in the queueing systems modelling servers are periodically monitored. The monitoring is possible due to appropriate construction of queueing systems models (compare subsection 2.2). *QL1*, *QL2*, and *QL3* places include the current values of queue's lengths. The queue's lengths are compared during *BALANCE* transition execution and *FEWEST* place acquires a number of the serwer which serves the fewest number of requests (the server with the shortest request queue). Guard functions associated to *T3*, *T4*, and *T5* transitions "open" (for the incoming requests) only this branch of the cluster which includes the least loaded server. The frequency of the queue's lengths measurement can be adjusted to derive the balance between additional system load caused by the measurement and the

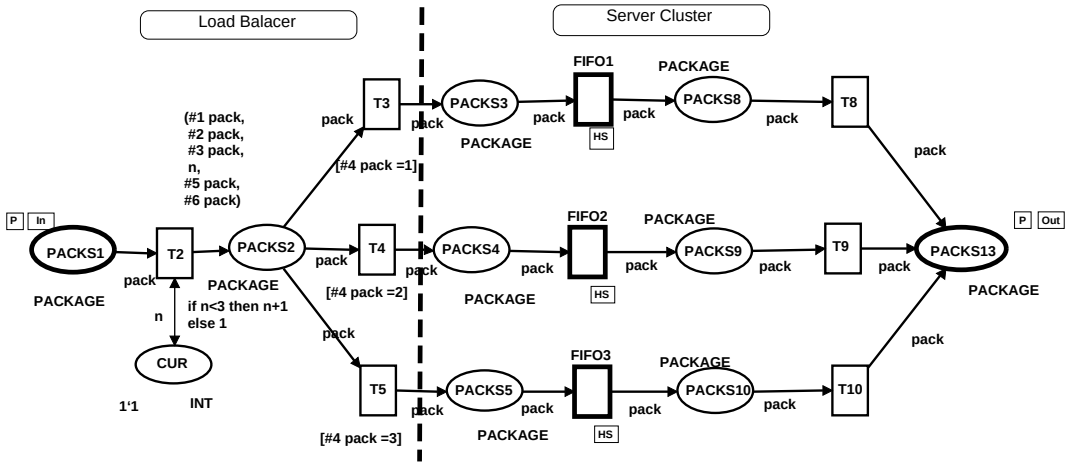


Figure 8. WWW cluster with round-robin load balancing policy

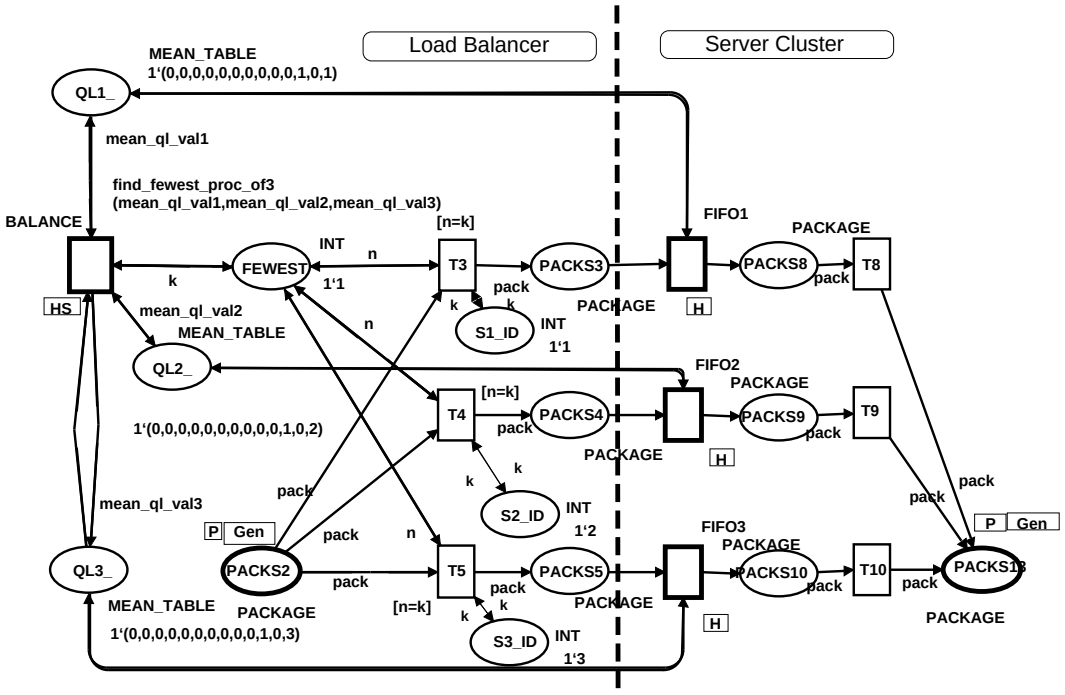


Figure 9. WWW cluster with fewest server processes first load balancing policy

quality of the balance process. It is possible to define digital filters to "smooth out" the queue length "signal".

3.3. Adaptive Load Sharing – Load Balancing Policy Model

Fig. 10 presents HTCPNs–based model of the computer cluster where the incoming Internet requests are redistributed among the cluster nodes according to adaptive load sharing load balancing policy inspired by [6], [10]. The model of the policy works as follows. During the system model execution lengths of the queues in the queueing systems modelling servers are periodically monitored. *QL1*, *QL2*, and *QL3* places include the current values of queue's lengths. During execution of the *BALANCE* transition the utilisation of each node of the cluster is calculated. Depending on the utilisation values some ranges of amounts of the Internet requests that each server may serve are calculated. The cal-

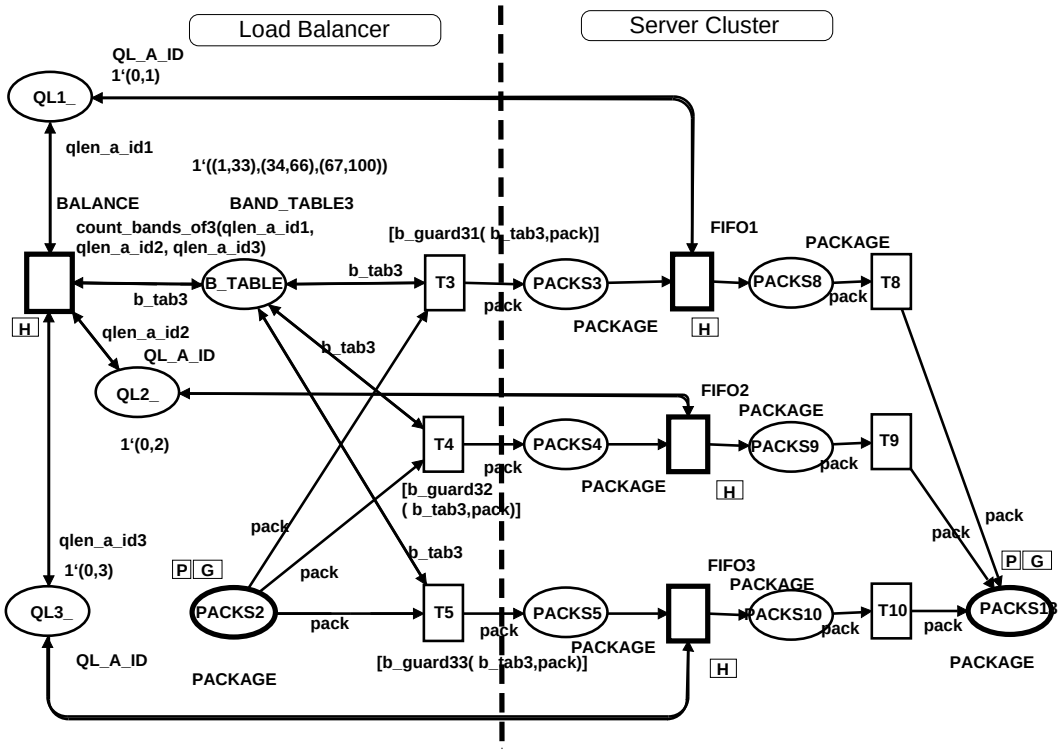


Figure 10. WWW cluster with adaptive load sharing load balancing policy

culated ranges are stored in *B_TABLE* place. Generally, servers having lower utilisation values will be given more chances to acquire the Internet requests in the future. Guard functions associated to *T3*, *T4*, and *T5* transitions can be understand as "valves" that adjust the amounts of the requests to be passed through to the servers according the range table stored in *B_TABLE* place. The frequency of the queue's lengths measurement can be adjusted to derive the balance between additional system load caused by the measurement and the quality of the balance process. It is possible to define digital filters to "smooth out" the queue length "signal".

3.4. Simulation Based on Load Balancing Policies Evaluation

For the above mentioned models of load balancing policies a set of simulation analysies was carried out. In Figure 11 results of 2 different simulations are shown. Figure 11 (left) includes queue lengths of 3 balanced servers where load balancing policy followed round-robin algorithm. In the (right) simulation a performance degradation of server 2 in 200000 time units was modelled. It can be easily seen, that the load balancer does not "notice" the performance degradation. The requests directed to second server are serviced later than the others.

The queue lengths for clusters where fewest server processes first and adaptive load sharing load balancing policies are coping with server 2 performance degradation are shown in Fig. 12. Both polices (fewest server processes first – Fig. 12 (left), adaptive load sharing – Fig. 12 (right)) "reconfigured" the loads for the cluster nodes and managed to keep the average queue lengths for all cluster nodes at the same level. The simulation experiment proved that dynamic load balancing polices better cope with dynamic changes during system execution. However, it must be noticed that the dynamic load balancing policies need some feedback information collected from the nodes of the cluster. To fulfill such requirement both modern load balancers and cluster nodes (e.g. WWW servers) software must be modified. Additionally the feedback data collection can increase the load of the system.

Figure 13 shows more possibilities for design of dynamic load–balancing policies. The simula-

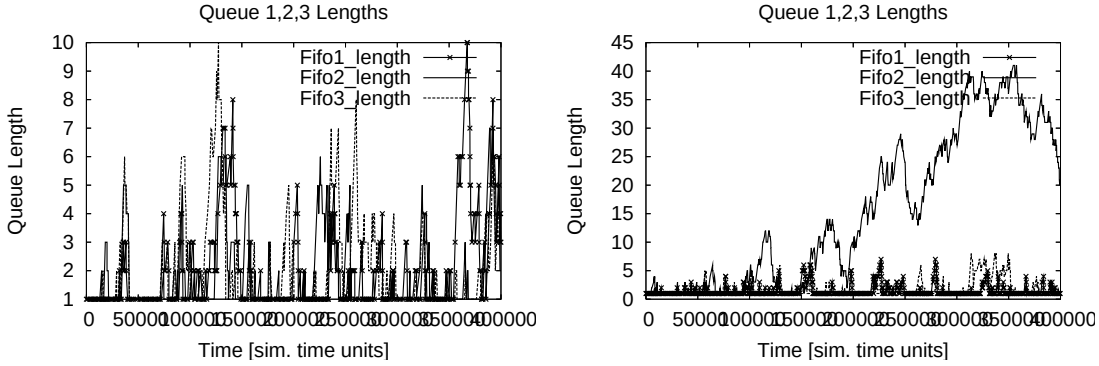


Figure 11. Queue lengths under round-robin load balancing policy (left) balanced (right) unbalanced after server 2 performance reduction

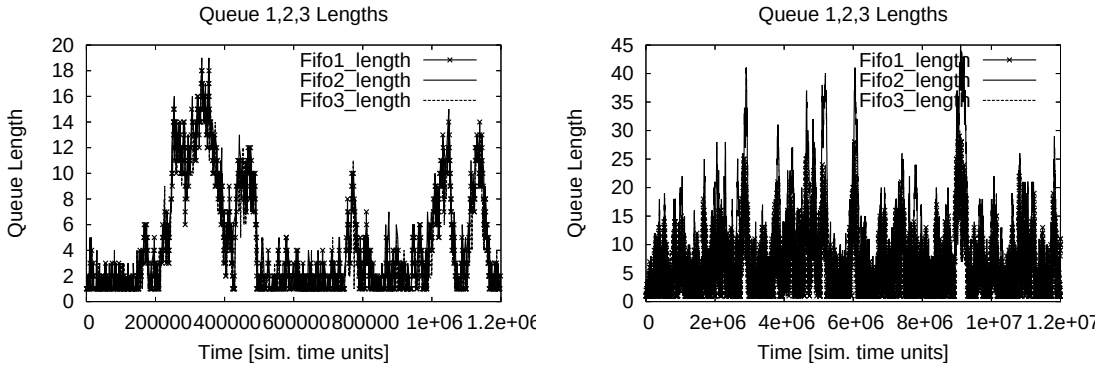


Figure 12. Queue lengths under fewest server processes first (left) and adaptive load sharing (right) load balancing policy after server 2 performance reduction

tion results show that the application of some feedback data to cluster state modification may cause system's behaviour similar to control–loop systems. In Figure 13 (left) the queue lengths oscillations caused by an inadequate data collection frequency may be noticed. The system in Fig. 13 (right) seems to "suffer" from the high sensibility that may in consequence lead to the instability.

4. Conclusions and Future Research

The first part of the paper introduces the HTCPNs–based software tool providing support for development and validation of web–server clusters executable models. The main concept of the tool lies in the definition of reusable HTCPNs structures (patterns) involving typical components of cluster–based server structures. The preliminary patterns are executable models of typical queueing systems. The queueing systems templates may be arranged into server cluster subsystems by means of packet distribution patterns. Finally, the subsystems patterns may be naturally used for top level system modelling, where individual substitution transitions "hide" the main components of the system. The final model is a hierarchical timed coloured Petri net. Simulation and performance analysis are the predominant methods that can be applied for the model validation. Queueing systems templates was checked whether they meet theoretically derived performance functions. The analysis of HTCPNs simulation reports enables to predict the load of the modelled system under the certain arrival request stream; to detect the stability of the system; to test a new algorithms for Internet requests redirection and for their service within cluster structures.

The second part of the paper includes the review of recently published research results con-

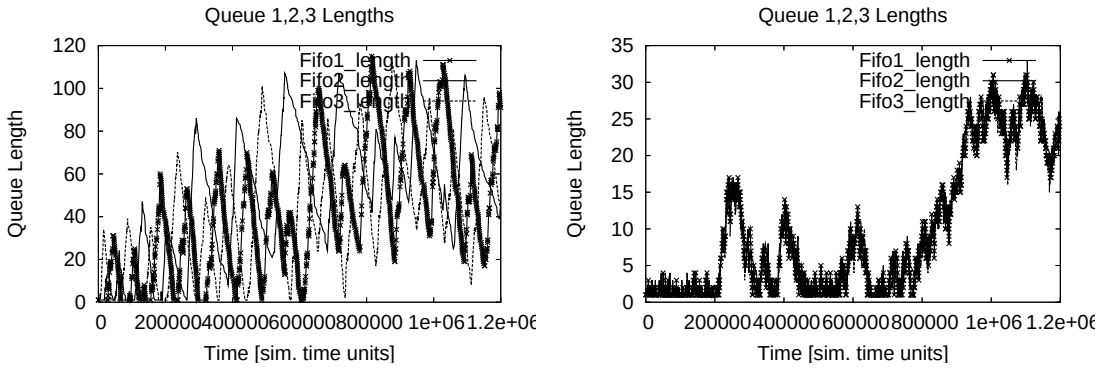


Figure 13. Queue lengths under fewest server processes first load balancing policy: queue lengths oscillation (left), high system sensitivity (right)

cerning application load balancing policies in Internet systems. Subsequently, the HTCPNs based models of some selected polices have been proposed. The most popular load balancing policies models such as "random", "round robin", and "fewest server processes first" as well as one experimental–"adaptive load sharing" have been applied and evaluated. The worked out HTCPNs structures become the integrated modules of the HTCPNs based software tool presented in the first part of the paper.

Currently, the software tool described in the paper can be applied for a limited web–server cluster structures modelling and validation. Thereafter the main stream of author's future research will concentrate on developing next web–server node structures models. This may result in following advantages. First, an open library of already proposed web–server cluster structures could be created and applied by the future web–server developers. Second, some new solutions for distributed web–server systems may be proposed and validated.

References

[1] F. Bause. Queueing Petri Nets – a formalism for the combined qualititative and quantitative analysis of systems. In *PNPM'93*, pages 14–23. IEEE, IEEE Press, 1993.

[2] H. Bryhni, E. Klovning, and O. Kure. A comparison of load balancing techniques for scalable web servers. *IEEE Network*, Volume 14(4):58–64, Jul./Aug. 2000.

[3] J. Cao, M. Andersson, C. Nyberg, and M. Khil. Web server performance modeling using an M/G/1/K*PS queue. In *CT 2003, 10th International Conference on Telecommunications*, pages 1501–1506. IEEE, 2003.

[4] V. Cardellini, E. Casalicchio, and M. Colajanni. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, Volume 34(2):263–311, June 2002.

[5] V. Cardellini, M. Colajanni, and P. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, Volume 3(3):28–39, May/June 1999.

[6] J. Guo and L. Bhuyan. Load balancing in a cluster-based web server for multimedia applications. *IEEE Transactions on Parallel and Distributed Systems*, Volume 17(11):1321–1334, 2006.

[7] K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use*, volume I-III. Springer, 1996.

[8] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, Volume 9(3-4):213–254, 2007.

[9] Y. Ji and I. S. Ko. A design of the simulator for web-based load balancing. *Springer LNCS*, Volume 4496/2007:884–891, July 2007.

[10] L. Kencl and J.-Y. L. Boudec. Adaptive load sharing for network processors. In *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, pages 545–554. IEEE, 2002.

[11] D. Kim, S. Lee, S. Han, and A. Abraham. Improving web services performance using priority allocation method. In *Proc. Of International Conference on Next Generation Web Services Practices*, pages 201–206. IEEE, 2005.

[12] S. Konunev. Performance modelling and evaluation of distributed component–based systems using Queuing Petri Nets. *IEEE Transactions on Software Engineering*, Volume 32(7):486–502, 2006.

[13] S. Kounev and A. Buchmann. SimQPN–a tool and methodology for analyzing Queueing Petri Net models by means of simulation. *Performance Evaluation*, Volume 36(4–5):364–394, 2006.

[14] M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured Petri Nets. *International Journal on Software Tools for Technology Transfer (STTT)*, Volume 2:98–132, 1998.

[15] T. T. Kwan, R. E. McGrath, and D. A. Reed. Ncsa's world wide web server: Design and performance. *Computer*, Volume 28(11):68–74, Nov. 1995.

[16] B. Linstrom and L. Wells. *Design/CPN Performance Tool Manual*. CPN Group, Univ. of Aarhus, Denmark, 1999.

[17] Linux virtual server project. http://www.linuxvirtualserver.org/.

[18] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh. Online response time optimization of Apache web server. In *IWQoS 2003: 11th International Workshop*, pages 461–478. Springer, 2003. LNCS.

[19] X. Liu, R. Zheng, J. Heo, Q. Wang, and L. Sha. Timing performance control in web server systems utilizing server internal state information. In *Proc. of the Joint Internat. Conf. on Autonomic and Autonomous Systems and International Conference on Networking and Services*, page 75. IEEE, 2005.

[20] loadbalancers.org. http://loadbalancer.org/.

[21] Meta Software Corporation. *Design/CPN Reference Manual for X-Windows*, 1993.

[22] G. Park, B. Gu, J. Heo, S. Yi, J. Han, J. Park, H. Min, X. Piao, Y. Cho, C. W. Park, H. J. Chung, B. Lee, and S. Lee. Adaptive load balancing mechanism for server cluster. *Springer LNCS*, Volume 3983/2006:549–557, May 2006.

[23] T. Rak and S. Samolej. Distributed internet systems modeling using tcpns. In *Proc. of International Multiconference on Computer Science and Information Technology*, pages 559–566. IEEE, 2008.

[24] S. Samolej and T. Rak. Timing properties of internet systems modelling using Coloured Petri Nets. In *Systemy czasu rzeczywistego – Kierunki badań i rozwoju*, pages 91–100. Wydawnictwa Komunikacji i Łączności, 2005. In Polish.

[25] S. Samolej and T. Szmuc. TCPN–based tool for timing constraints modelling and validation. In *Software Engineering: Evolution and Emerging Technologies, Volume 130 Frontiers in Artificial Intelligence and Applications*, pages 194–205. IOS Press, 2005.

[26] S. Samolej and T. Szmuc. Time constraints modeling and verification using Timed Colored Petri Nets. In *Real–Time Programming 2004*, pages 127–132. Elsevier, 2005.

[27] S. Samolej and T. Szmuc. Dedicated internet systems design using Timed Coloured Petri Nets. In *Systemy czasu rzeczywistego – Metody i zastosowania*, pages 87–96. Wydawnictwa Komunikacji i Łączności, 2007. In Polish.

[28] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, Volume 23(5):9–18, 20–21, May 1990.

[29] T. Schroeder, S. Goddard, and B. Ramamurthy. Scalable web server clustering technologies. *IEEE Network*, Volume 14(4):38–45, May/June 2000.

[30] Z. Shan, C. Lin, D. Marinecu, and Y. Yang. Modelling and performance analysis of QoS–aware load balancing of web–server clusters. *Computer Networks*, Volume 40:235–256, 2002.

[31] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. Wiley-IEEE Computer Society Press, April 1995.

[32] F. Spies. Modeling of optimal load balancing strategy using queueing theory. *Microprocessing and Microprogramming*, Volume 41:555–570, 1996.

[33] Thomas-kern load balancers. http://www.thomas-krenn.com.

[34] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. Analytic modeling of multitier Internet applications. *ACM Transactions on the Web*, Volume 1(2), 2007.

[35] L. Wells. Performance analysis using CPN tools. In *Proc. of the 1st Inter. Conf. on Performance*

Evaluation Methodolgies and Tools, 2006. Article No. 59.

[36] L. Wells, S. Christensen, L. Kristensen, and K. Mortensen. Simulation based performance analysis of web servers. In *Proc. of the 9th Internat. Workshop on Petri Nets and Perf. Models*, page 59. IEEE, 2001.

[37] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo. Workload-aware load balancing for clustered web servers. *IEEE Transactions on Parallel and Distributed Systems*, Volume 16(3):219–233, March 2005.

[38] W. Zhang. Linux virtual server for scalable network services. In *Ottava Linux Symposioum. Proceedings*, 2000.

[39] Z. Zhang and W. Fan. Web server load balancing: A queueing analysis. *European Journal of Operational Research*, Volume 186(2):681–693, April 2008.

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

e-Informatica

ISSN 1897-7979