Transformation from Requirements to Design for Service Oriented Information Systems[1]

Lina Ceponiene[2] and Lina Nemuraite[2]

[2] Kaunas University of Technology, Studentu 50-308,
LT 51368 Kaunas, Lithuania
kavalina@soften.ktu.lt, lina.nemuraite@ktu.lt

Abstract. Service-Oriented Architecture (SOA) and Web services are becoming the universally accepted architectural style for development of modern information systems of enterprises. But the methods of design in SOA are not well established yet. The most of current methodologies are focused on composition of business processes from services. In this work, SOA based design is considered as design of information system where modelling of services and processes composed of services is related to modelling of entities comprising service execution context. It is demonstrated, that various forms of UML 2.0 interactions and state machines fit well for representation of SOA related concepts – services, protocols, choreography, orchestrations, and transactions. The proposed design method consists of two steps – making comprehensive specification of requirements and transforming it to design using State Coordinator pattern that enables loose coupling of stateless services into system operating on the base of information about states of entities.

1 Introduction

Service-Oriented Architecture (SOA) and Web services [22, 27] are becoming the universally accepted architectural style for development of information systems of enterprises. As foremost Web services have arisen as technological challenge in late 1999, methods of design of service-oriented systems are not well established up till now. Existing modelling approaches such as Object-Oriented Analysis and Design (OOAD), Software Component Based Design (CBD), Enterprise Architecture (EA) frameworks, and Business Process Modelling (BPM) have provided high-quality practices for development of enterprise information systems. But for development of service oriented systems, as stated in [31], more advanced techniques are required.

What are features of service orientation that do not fit to familiar methodologies? A service is an operation offered as an interface that stands alone in the model, without encapsulating state, as entities and value objects [15]. Though concepts of services have arisen from technical frameworks, in service-oriented design definition of service must be originated from business domain, not from technology. Unlike enti-

[1] The work is supported by Lithuanian State Science and Studies Foundation according to Eureka programme project "IT-Europe" (Reg. No 3473)

ties, service is described in terms of what it can do for a client; it has interface, specifying set of operations, and makes contract with its client, so defining responsibilities to fulfil this interface.

The exclusive feature of service-oriented design is the separation between behavioural objects, i.e. services, and persistent entities (information objects), in contrast with object oriented design. A good service design has three characteristics [15]: its operations correspond to domain concepts that are not the natural parts of entities or value objects; the interface is defined in terms of other elements of the domain model; the operations are stateless. Statelessness of service means that it is independent of context, and any client can use any instance of a particular service without regard to the history of this instance. The execution of a service uses external information, and may change that information. But the service does not hold the state of its own that affects its own behaviour, unlikely the most domain objects.

In reality, the use of services is dependent on rules of business processes in hand. These rules are often expressed in terms of states of information entities comprising service execution context. In this work, design of information systems embracing services of business domain is considered, and the State Coordinator pattern is proposed for loose connection of stateless services into system that operates on the base of information about persisted states of entities. SOA based design is considered as design of information system where modelling of services and processes composed of services is related with modelling of entities comprising context. This differs from majority of proposed techniques, where emphasis is made on modelling of business processes but information modelling is limited to definition of types of messages and variables [1], textual notes [8], or not considered at all.

The proposed design method consists of two steps – making comprehensive specification of requirements (Design Independent Model (DIM) [10, 11]), and transforming requirements to design – Platform Independent Model (PIM) in MDA terminology [18]. It is demonstrated, that various forms of UML 2.0 [25] interactions and state machines fit well for representation of SOA related concepts – services, protocols, choreography, orchestration, and transactions. Secondly, it is shown that DIM specification (using OCL [26, 28] and principles of contract-based design [12]) makes it possible to formalize transformation from requirements to design.

The paper is structured as follows: in Section 2 service concepts and related work is characterized. In Section 3 the principles of requirements specification for service-oriented design are presented and illustrated with the example. In Section 4 service design pattern is proposed. In Section 5 transformation from requirements to design is described. Finally, Section 6 makes conclusion and reasoning about future work.

2 Service Concepts and Related Work

Related approaches for modelling in SOA are associated with Business Process Modelling Languages BPML [2], BPMN [8], BPEL [1], WSCI [3], WS-CDL [17]; standards for Business Transactions [9], Unified Modelling Methodology (UMM) [24] and ebXML [14]; the most popular implementation language is BPEL, or BPEL4WS; relationships of these languages to Web Services Description Language (WSDL) [29]

as well as generation of WSDL specifications from designs of services are well established.

Design of services deals with three levels of abstraction: operations, services (groupings of operations) and business processes. Operations represent atomic business transactions (logical units of work). Execution of an operation usually causes one or more persistent data records to be read, written, or modified, and additional operations may be invoked. Service corresponds to class concept. For design of operations of the service, object (e. a. [19]) or component-oriented (e. a. [12]) methods are suitable, with the particularity, that services are pure behavioural concepts.

Business processes represent long running flows of actions and activities performed in order to respond to business events, and to achieve business goals. Business processes require multiple invocations of business internal services and services rendered by external business systems. The rules of sequencing of message exchange patterns of business-to-business collaborations across business process are termed as process choreography. Besides choreography concept that is used for definition of business-to-business processes, the concept of orchestration is essential for modelling of internal business processes serving for execution of interactions that particular process can manage. Concepts of choreography, orchestration and multiparty business transactions are extensively used in Web services literature; the intelligible clarification of terms may be found at EBPML Web site (e.g. [13, 22]).

The goal of service-oriented design is systematic construction of operations, services, and organised sets of services. Many works are devoted to development of business processes, composed of services; composition rules and phases [30]; emerging W3 Consortium and OASIS standards are concerned with choreography, orchestration, transaction, context and coordination frameworks. In current business process modelling languages, devoted for composition of services, design of business processes is not integrated with design of services themselves. Similarly, object-oriented and component-based methods, suitable for design of operations and services, are lacking of service composition potential. In UMM, design of services is linked with design of global business processes (choreographies), but orchestration is not considered and services are not integrated with entities of domain model; so this methodology is also insufficient for end-to-end development of service systems.

In this work, system of services is constructed, rather than single business process, and development process is considered going from requirements to code. Specification of choreographies and orchestrations of business processes is based on UML 2.0 interactions and, specifically, interaction overview diagram that represents fruitful combination of activity and sequence diagrams whereas established methods are based on activity diagram-like representations or using activity and sequence diagrams alternately.

For execution of services, transition systems semantics and state machines mechanisms are universally accepted (e.g. [5, 7, 4]), where states usually represent persistent states observable in business domain. In our work, both persistent states and behavioural states (performing actions or waiting for events) are taken into account, and state machines of services are interrelated with state machines of entities. During execution, system operates as composite state machine, where transitions are fired by external events (received messages about requests of services) and restricted by per-

sisted states of entities. Transition rules coincident with service usage contracts are separated from services; they may be implemented using rule checking operations or stored in rule base, so design may be flexible to possible changes in business domain.

The proposed transformation is based on State Coordinator pattern that is somewhat similar to combination of classical Façade and State patterns [16]. State Coordinator serves as front end for receiving service request messages (as Facade), and makes choice of operation for execution subject to context (as State). Additionally, it takes into consideration interactions between services. For SOA design, many of classical patterns are reused [16], and service-specific patterns are proposed [5, 23, 25], but service composition mostly is based on Business Process Modelling Languages. State Coordinator may be a simple variation of Business Process execution engine that may be used for customary development as much as for model driven design.

3 Requirements definition

In this section, principles of specification of requirements relevant for intended goals to formalize service-oriented design are presented. Detailed model of requirements must define overall state and behaviour of intended information system independently of future design. Requirements definition consists of two phases: initial requirements and system requirements.

Initial requirements are described informally using Use Case diagrams and Use Case templates that are filled using terms from domain model. Every step of use case is described as user interaction with the system using pre and post conditions. To be precise, domain and use case model are constructed simultaneously: during use case analysis every time when new object types are discovered domain model is updated.

In second phase, initial requirements are further evolved. Every use case is transformed into interface between the user and the system, capturing interactions between (possibly external) interfaces, and every use case step is transformed to operation; use cases are detailed using sequence diagrams, where operations are specified in OCL. Initial use case diagram and DIM of illustrative Publication Agency are presented in Figure 1; sequence diagrams representing interactions between user and system during execution of single use case (Submit) is presented in Figure 2, together with specification of operation. Two kinds of sequence diagrams are used for use cases: interaction between two participants (Business interaction protocol that may be represented by protocol state machine) and namely interaction protocol that may be represented by interaction (Business transaction) state machine. Protocol state machines are introduced in UML 2.0, but interaction state machines are not considered. Sometimes they may coincide with port state machines [20] but in general port may be designed for collection of interactions.

The interactions and patterns of interactions between participants of business process represent choreographies of this process executed using services; the process of internal coordination of all interactions performed in the system of individual participant makes the orchestration. State Coordinator pattern proposed in this paper may be considered as a kind of orchestration engine.

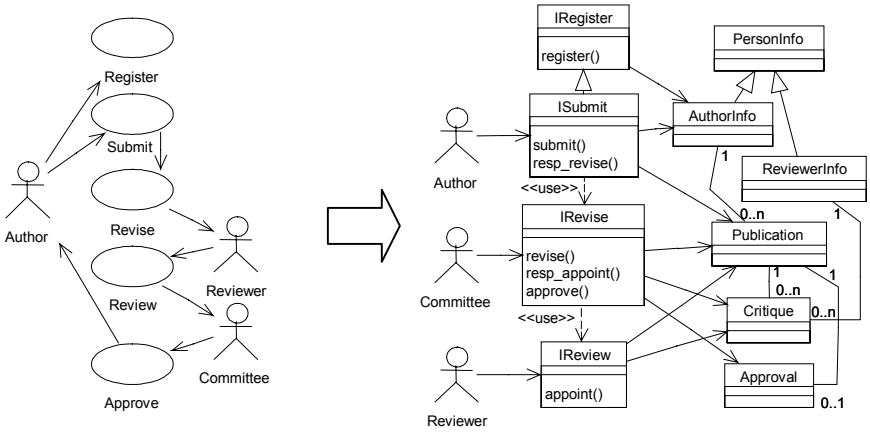


Fig. 1. Initial requirements (Use Cases) and requirements specification (DIM)

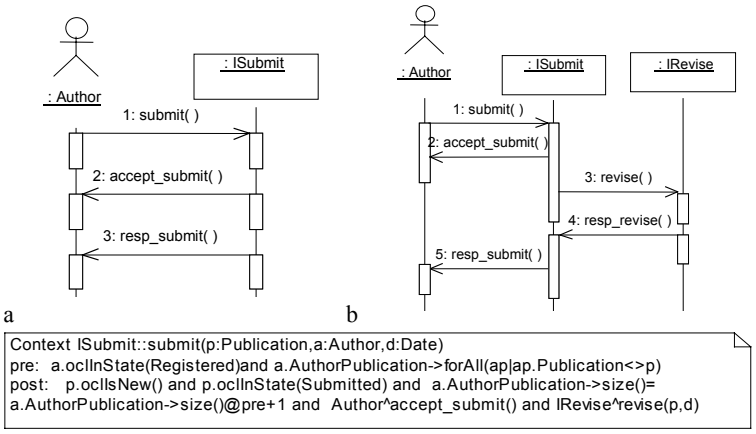


Fig. 2. Actor/Interface interaction (a) and Interface interaction (b)

Sequence diagrams like Figure 2a, representing client viewpoint, are not sufficient for comprehensive specification of requirements, because realization of use case may require usage of other services supported by the intended system, or other business systems. Both offered and required interfaces are captured in interaction sequence diagrams like Figure 2b. In Figure 3, interaction fragments represent choreography of global business process “Submission” (for illustrative purpose, suppose that Isubmit, IRevise and IReview are interfaces of different business systems, and “Revise” is automatic service).

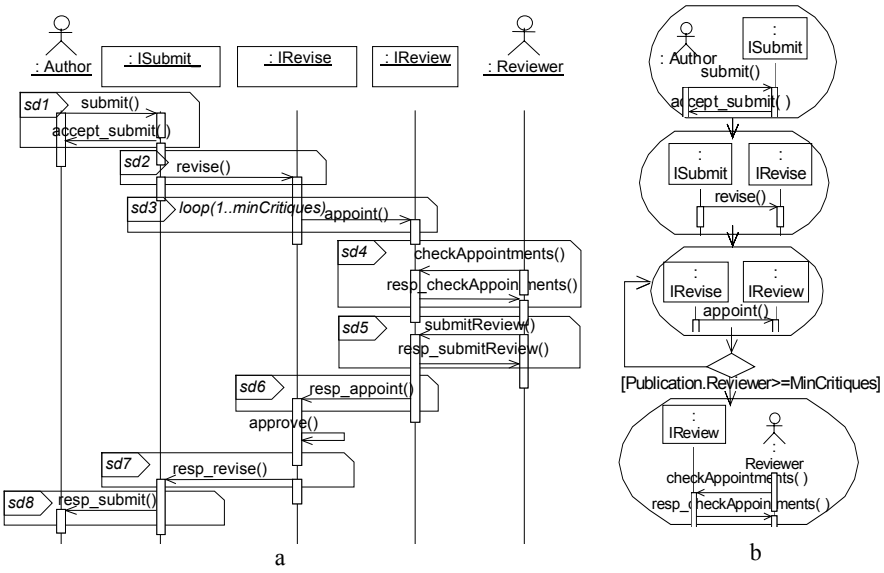


Fig. 3. Interaction fragments (a) and interaction overview (b) representing choreographies

For reconciliation of DIM, all interactions are transformed to state machines where states of the system are represented by states of interfaces and entities of domain model (Fig. 4); composite state machines render compound business transactions.

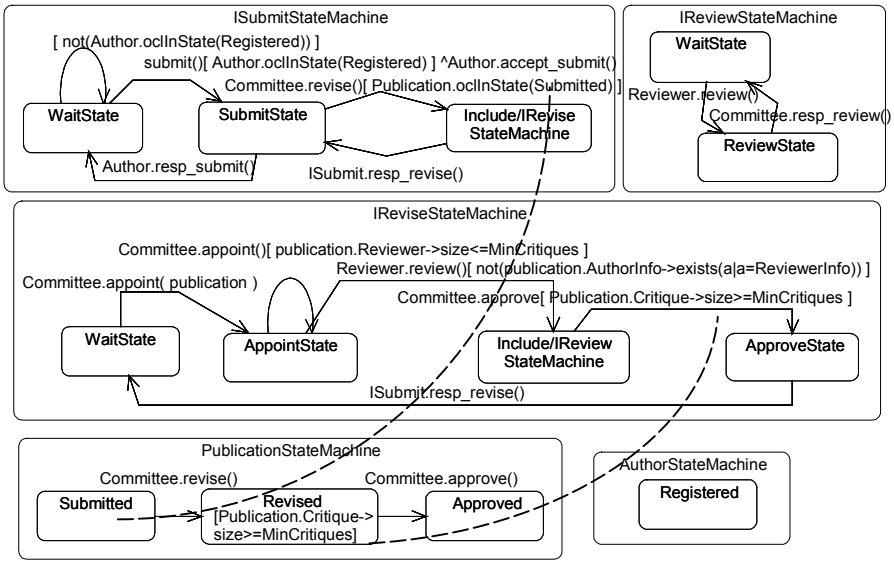


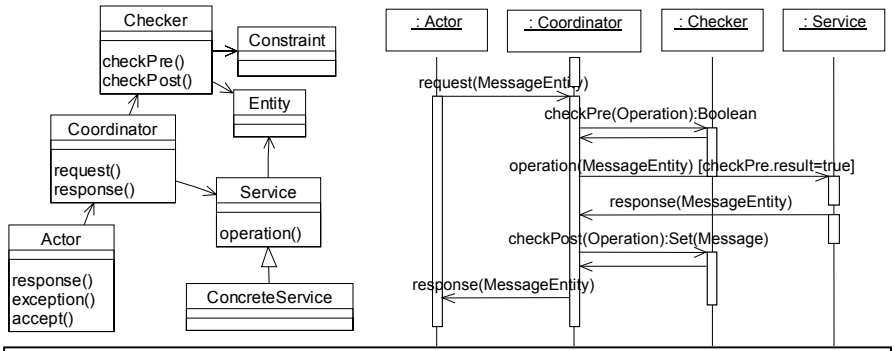
Fig. 4. Concordance between state machines of interfaces and entities

State machine is the next-to-last requirements modelling step that may be performed semi-automatically with support of CASE tool [10]. During this step different interac-

tion scenarios are consolidated and converted to class diagram like the one in Figure 1 (detailed specification is not presented due to limits of space).

4 State Coordinator pattern

Transformation from design independent model to design (PIM) consists of several steps and may be done in several ways. Mapping interfaces to services results in coarse design, and mapping operation constraints to methods is in responsibility of detailed design. Here we are considering architectural design, during which elements of specification are allocated to realizing architectural elements. For service-oriented design, the State Coordinator pattern is proposed (Fig. 5). The Coordinator handles incoming messages that may be of two types: requests for some service operation and response from the service about operation execution results. In Figure 5, Coordinator's reaction to received messages is presented graphically using sequence diagram and specified in OCL as post-conditions of Coordinator's operations.




```
Context Coordinator::request(m:Message)
post:
let pr:oclMessage=Checker^checkPre(m.Operation) in pr.hasReturned()
 and if pr.result()=true then m.Operation.Service^m.Operation(m)
     else Actor^exception(m.Operation.ExceptionMessage) endif
Context Coordinator::response(m:Message)
post:
let ps:oclMessage=Checker^checkPost(m.Operation) in ps.hasReturned()
 and ps.result()→ forAll(msg|let op: Operation = msg.Operation in
    if msg.receiver.oclisKindOf(Interface) then
     let pr:oclMessage=Checker^checkPre(op) in pr.hasReturned() and
       if pr.result()=true then op.Service^op(op.RequestMessage)
       else Actor^exception(op.ExceptionMessage) endif endif
    if msg.receiver.oclIsKindOf(Actor) then
     if op.clIsKindOf(Acceptance)
      then Actor^acceptance(op.Request.AcceptanceMessage)
      else Actor^response(op.Request.ResponseMessage)
    endif
```

Fig. 5. State Coordinator pattern and its principle of working

On received request, State Coordinator handles message, unfolds the name of requested operation and calls checker to check precondition of that operation. Precondi-

tions and post conditions are specified in Constraint base using attributes and relationships of entities from domain model. If Checker returns “false”, the rejection message is sent. If Checker returns “true”, Coordinator calls operation and service returns response about delivery or acceptance of service. Before sending response to requestor, Coordinator asks Checker to check message expressions specified in post conditions and possibly returns the set of messages that must be sent to internal or external services to fulfil the request. It may be zero, one or more messages that must be sent sequentially, in parallel or broadcast. If there are no messages to send, Coordinator resends response to the requestor (as shown in Fig.5). Otherwise, it sends messages to internal or external services as specified in message expressions, and treats received responses in the same way as before.

Message expressions mostly denote sending of message to successive interface, but they may express sequence of messages, messages sent in parallel to different targets (messages joined with “and”) or even multicast message flow with dynamic targets. Indeed, every kind of these interactions may be described in OCL. If responses must be received during operation execution it means that there is composite operation, and every received message presents different operation described with pre and post conditions. In other words, every request, acceptance or response is treated as separate operation what significantly streamline reasoning. Using message expressions in post conditions, services may be composed to the system in the recursive way as services secondarily called may have calls to other services, and so on. The implementation of Checker and Constraint may vary from direct checking operations to complex rule checking engines and repositories.

The purpose of Coordinator is the same as of other design patterns [16]: to “normalise” behaviour, discovering recurring activities and concentrating them in separate classes thus making the cohesive units of behaviour. In compound Web services environment, such recurring behaviour is receiving/sending of messages, checking context and selecting services for execution. State Coordinator pattern was constructed on the base of Facade and State patterns, as it was no suitable Web Service pattern for this purpose [5, 23, 25]. It is obvious, that for practical development considerably more patterns should be used. In large systems, coordinator may be attached to every composite service.

State Coordinator pattern is simple alternative for Business Process execution engine. Coordinator handles incoming message and passes it to services according to its actual context. Coordinator uses the assistance of Checker that checks constraints (pre-conditions and post-conditions) of operations. According to the principles of good SOA design, operations of services must be stateless. Information about states is captured by entities, and all constraints describing services subject to state changes are kept in Constraint base.

Coordinator may interact with external services or own composite services but nevertheless they are treated as stateless services. It deals with constraints and signatures of operations described by constraints that represent logic of usage of these operations in Information System, and selects concrete service to fulfil request or sends response about inability to do this. If new services are inserted or business rules are changed, constraints must be supplemented or modified.

There are many ways to proceed from requirements to design though we believe that it would be valid to use State Coordinator in SOA related design when business process execution language is not used. Resulting design may be implemented using J2EE, MS .Net or other framework with message-oriented middleware, creating operations for checking constraints. Checker also may be thought as some kind of rule checking component when rules are stored declaratively in rule base.

5 Transformation to design

Transformation from DIM to PIM, based on State Coordination pattern, is presented in Figure 6. Transformations concerning detailed design and implementation are not considered in this design phase. Signatures and body conditions of PIM operations are obtained from body conditions of DIM query operations or post-conditions of non-query operations (except of message expressions that together with preconditions are allocated to constraints). For meaningful design, operations in DIM must have been discovered in the way ensuring right division of responsibilities, and description of post-conditions must hold all information for design of methods implementing behaviour.

The transformation from DIM to PIM is based on mapping between elements of meta-models. This mapping is described by the set of rules that define how the elements of the source model (DIM) are allocated to elements of the target model (PIM). Main elements of DIM and PIM meta-models related by transformation rules are depicted in Figure 6. These rules are specified using simple transformation language based on OCL [18, 28]; the main transformations are shown in Figure 7, hiding details how every relationship, attribute, association end, etc. is transformed.

Transformations using State Coordinator pattern mainly are straightforward:

- Coordinator service and abstract Actor class are created in PIM (transformations DIM2Coordinator and DIM2Actor);
- DIM entities are transformed to PIM entities (transformation Entity2Entity);
- DIM interfaces are transformed to PIM services with interfaces (transformations Interface2Service and Interface2ServiceInterface);
- DIM operations are transformed to PIM operations. All the parameters of each DIM operation are allocated to one parameter (of type MessageEntity) of PIM operation (transformation Operation2Operation);
- PIM message entities (MessageEntity) (also called value objects) are created for DIM request, response, acceptance operations and operation preconditions. The parameters of the operation (object types and data types from model of problem domain) are transformed into elements (MessageElement) of message entities (transformation Operation2Messages);

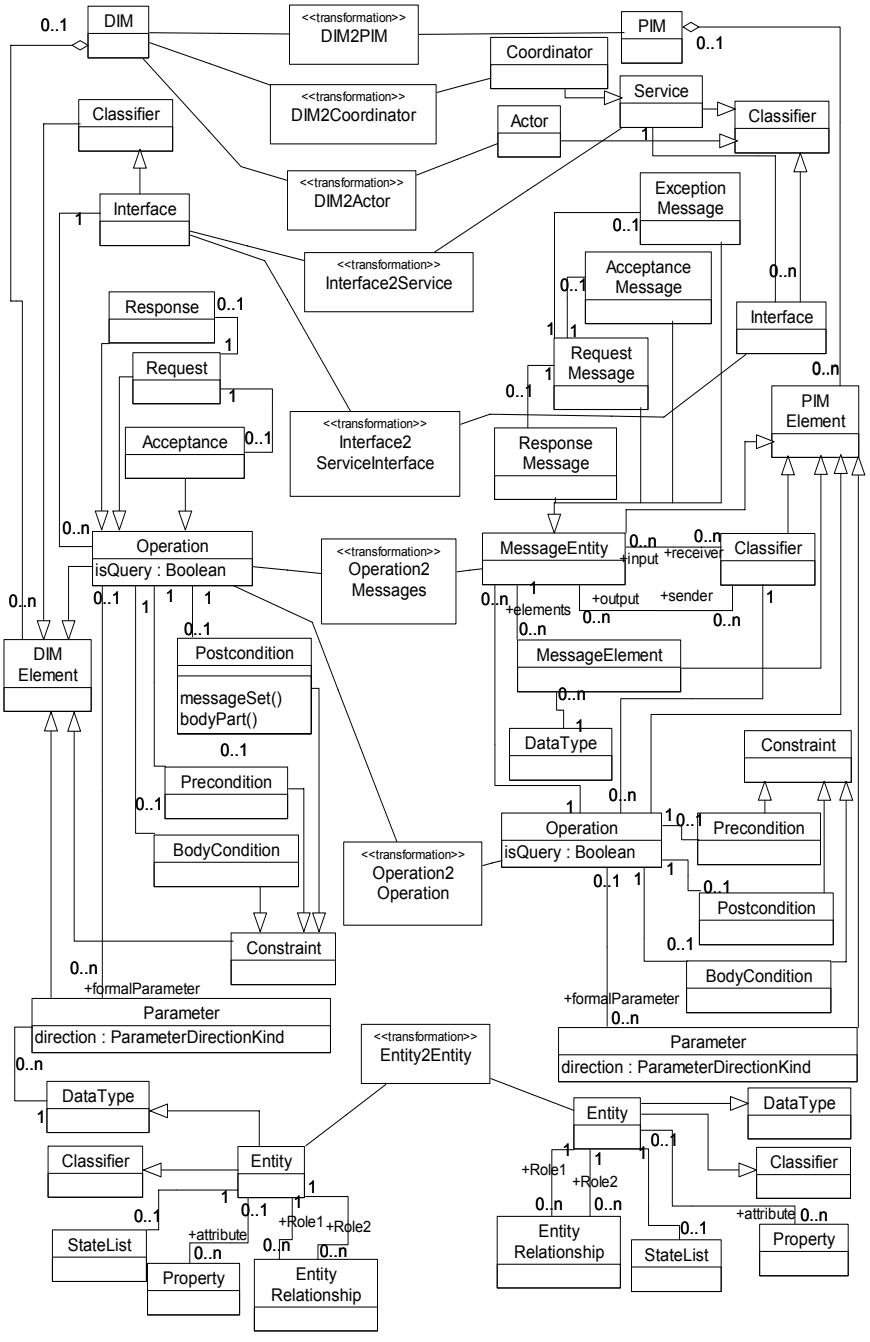


Fig. 6. Transformation from requirements model (DIM) to design (PIM)

```
Transformation DIM2PIM (UML, UML)
 {source DIM : UML :: Package;
 target PIM : UML :: Package;
source condition DIM.Interface→ notEmpty();
target condition DIM.name=PIM.name;
bidirectional;
mapping
 try Interface2Service on DIM.Interface <~> PIM.Service;
 try Entity2Entity on DIM.Entity <~> PIM.Entity;
 try DIM2Actor on DIM <~> PIM.Actor;
 try DIM2Coordinator on DIM <~> PIM.Coordinator;}
```

```
Transformation Interface2Service(UML,UML)
{source int:UML:Interface;
 target s:UML:Service;
target condition s.name=int.name.concat('Service');
bidirectional;
mapping
 try Operation2Operation on int.Operation <~> s.Operation;
 try Interface2ServiceInterface on int <~> s.Interface}
```

```
Transformation Operation2Operation(UML,UML)
{source opDIM:UML::Operation;
 target opPIM:UML::Operation;
target condition
 opPIM.name=opDIM.name and opPIM.Precondition=opDIM.Precondition and
 opPIM.Bodycondition = opDIM.Bodycondition and
 opPIM.Postcondition= opDIM.Postcondition and
 opPIM.formalParameter→ exists(m| m.oclIsKindOf(MessageEntity);
bidirectional;
mapping try Operation2Messages on opDIM <~>opPIM.MessageEntity}
```

```
Transformation Operation2Messages(UML,UML)
{params req:UML::RequestMessage;   resp:UML::ResponseMessage;
        xcp:UML::ExceptionMessage; accp:UML::AcceptanceMessage;
source op:UML::Operation;
target msg:UML::Set(MessageEntity);
target condition
 req.name=op.name.concat('ReqMsg') and
 req.elements=op.formalParameter→ iterate(p;acc:Set(MessageElement)|
       if p.direction=in or p.direction=inout then
        let elem:MessageElement=p in acc.including(elem) endif) and
 msg.including(req) and resp=req.ResponseMessage and
 resp→ notEmpty() and resp.name=op.name.concat('RespMsg') and
 resp.elements=op.formalParameter→iterate(p;acc:Set(MessageElement)|
      if p.direction=out or p.direction=inout then
       let elem:MessageElement=p in acc.including(elem) endif) and
 msg.including(resp) and
 if op.Precondition→ notEmpty then
  excp=req.ExceptionMessage and excp→ notEmpty and
  excp.name=op.name.concat('ExcpMsg') and
  let pr:MessageElement in pr.oclIsTypeOf(Constraint) and
   pr=op.Precondition and excp.elements=req.elements.including(pr)
  and msg.including(excp) endif and
 if op.Acceptance→ notEmpty() then
  req.AcceptanceMessage=accp and accp→ notEmpty and
  accp.name=op.name.concat('AccpMsg') and accp.elements=req.elements
  and msg.including(accp) endif;
 bidirectional;}
```

Fig.7. Part of transformations from DIM to PIM

- Preconditions and message parts of post conditions of DIM operations are transformed to preconditions and post conditions of PIM operations; body conditions

(of post conditions) of DIM operations are transformed to body conditions (of methods) of PIM operations (this transformation (Constraint2Constraint) is not shown on the Figure 6 as soon as other details). The ultimate design of methods is deferred to the phase of detailed design; the implementation of methods sometimes may be achieved by direct transformation from OCL to program code, sometimes it must be fulfilled manually.

In presented transformation some assumptions were made that may vary in different circumstances, for example, concrete naming scheme. Also, all preconditions here have textual descriptions, and exception messages are created by concatenation of negation of precondition and standard textual phrase. In practice, requirements for message entities may be predefined in requirements phase.

The implementation of transformations from DIM to PIM as extensions of UML CASE tools may be another topic of research. There are many alternatives to choose of:

- To base transformations on MOF to XML mapping (Metadata Interchange (XMl) – OMG specification of standard model transfer format), but there are difficulties raised by incompatibility of different implementations of different XMI versions by CASE tools vendors.
- Other alternative is to base on MOF to Java mapping − Java™ Metadata Interface (JMI) created by Sun Microsystems. It is platform independent dynamic infrastructure for metadata creation, storage, interchange and management. But dependency on application programming interface (API) of CASE tool remains unresolved.
- There are many other similar mappings and repositories with analogous problems.
- Eclipse Modelling Framework supports several metadata management scenarios and seems the most promising solution capable to sustain compatibility for different functionalities of CASE tools. Transformations in such a case may be implemented as Eclipse plugins.

Trial implementations of transformations, concerned with this work, were made (and are under further development) as extensions to UML CASE tools Argo UML (using native API) and Magic Draw (using JMI and API). Though our objectives are to propose conceptual solution for going from requirements to design and implementation serves only as demonstration of its validity, in the future the implementation issues should be more deeply concerned focusing on Eclipse Modelling Framework.

6 Conclusion

Behaviour has many forms that must be modelled during requirements definition phase for subsequent service-oriented design: party interaction protocols; choreographies and orchestration of business processes, and transactions; interfaces and interface interactions; entities, operations and constraints. It is demonstrated that all of these concepts may be defined, analysed, and reconciled in Design Independent Modelling, where possibilities of UML 2.0 and OCL are employed.

The main purpose of the work was to demonstrate that comprehensive definition of requirements of Information System enables to obtain meaningful design in formal way. As result, transformation from requirements to architectural design is presented,

during which elements, defined in requirement specification, are allocated to design elements following State Coordinator pattern proposed for service-oriented design of Information Systems.

Resulting design may be further subjected to detailed design of operations and transformed to implementation in WSDL and web services framework. Proposed pattern is simple alternative for development of service-oriented information systems when business process execution languages are not used.

References

1. Andrews, T. et all: Business Process Execution Language for Web Services Version 1.1. (2003) http://www-128.ibm.com/developerworks/library/specification/ws-bpel
2. Arkin, A.: Business Process Modeling Language. BPMI.org. (2002) http://www.bpmi.org
3. Arkin, A. (ed.): Web Services Choreography Interface (WSCI) 1.0. BEA Systems, Intalio, SAP, Sun Microsystems (2002) http://www.w3.org/TR/wsci
4. Baina, K., Benatallah, B., Casati, F., Toumani, F.: Model-Driven Web Service Development. In: A.Persson, J.Stirna (eds.): Advanced Information Systems Engineering. 16th international Conference, CaiSE 2004. Riga, Latvia, June 7-11, 2004, Springer-Verlag Berlin Heidelberg New York (2004) 290-306
5. Benatalah, B., Dumas, M., Fauvet, M.C., Rabhi, F.A., Sheng, Q.Z.: Overview of some patterns for architecting and managing composite web services. ACM SIGecom Exchanges archive, Vol. 3, Issue 3, Summer, 2002 (2002) 9-16
6. Benatallah, B., Casati, F., Toumani, F., Hamadi, R.: Conceptual Modeling of Web Service Conversations. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds.): Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), LNCS vol. 2681, Springer Verlag, Klagenfurt, Austria (2003) 449-467
7. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: A foundational vision of e-services. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds). Proc. of the CAiSE 2003 Workshop on Web Services, LNCS, vol. 3095 (2003) 28-40
8. Business Process Modelling Notation (BPMN). Version 1.0 – May 3 2004, BPMI.org. (2004) http://www.bpmn.org
9. Business Transaction Protocol Primer. Organization for the Advancement of Structured Information Systems (2002) http://www.oasis-open.org
10. Ceponiene, L., Nemuraite, L.: Design Independent Modeling of Information Systems Using UML and OCL. In: Barzdins, J. et all, (eds.): Databases and Information Systems, Computer Science and Information Technologies, Vol. 672. Sixth International Baltic Conference on Data Bases and Information Systems (DB&IS'2004), Riga, Latvia (2004) 357-372
11. Ceponiene, L., Nemuraite, L., Paradauskas, B.: Design of schemas of state and behaviour for emerging information systems. In: Thalheim, B., Fiedler, G. (eds.): Computer Science Reports, Vol. 14. Branderburg University of Technology at Cottbus (2003) 27–31
12. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML. The Catalysis Approach. Addison Wesley, Boston (1999)
13. Dubray, J. J.: A new model for multiparty collaborations. EBPML.org, (2002) http://www.ebpml.org
14. ebXML Business Process Specification Schema. Version 1.01. Business Process Project Team, UN/CEFACT and OASIS (2001) http://www.ebxml.org/specs
15. Evans, E.: Domain Driven Design. Tackling complexity at the heart of software. Addison-Wesley, Boston (2003)

16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Pearson Education (1994)
17. Kavantzas, N. (ed), Olsson, G., Mischkinsky, J., Chapman, M.: Web Services Choreography Definition Language. Oracle Corporation (2003)
18. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison Wesley, Boston (2003)
19. Mellor, S.J., Balcer, M.J.: Executable UML. A foundation for model-driven architecture. Addison-Wesley, Boston (2002)
20. Mencl, V.: Specifying Component Behavior with Port State Machines. Electronic Notes in Theoretical Computer Science 101 (2004) 129–153
21. Monday, P.B.: Web Service Patterns: Java Edition. Springer-Verlag New York, Inc. (2003)
22. Papazoglou, M. P., Dubray, J.: A Survey of Web service technologies. Technical Report DIT-04-058, Informatica e Telecomunicazioni, University of Trento (2004)
23. Singh, I., Brydon, S., Murray, G., Ramachandran, V., Violleau, T., Stearns, B.: Designing Web Services with the J2EE™ 1.4 Platform JAX-RPC, SOAP, and XML Technologies. Addison Wesley, Boston (2004)
24. UN/CEFACT Modeling Methodology. UNCEFACT/TMWG. (2002) http://www.unece.org/cefact/umm/ umm_index.htm
25. Unified Modeling Language Superstructure Specification. Version 2.0, OMG document ptc/03-08-02 (2003) http://www.omg.org
26. Unified Modeling Language: OCL Version 2.0. OMG document ptc/03-08-08 (2003) http://www.omg.org
27. W3 Consortium. Web Services Architecture. W3C WG (2004) http://www.w3.org
28. Warmer, J.B., Kleppe, A.G.: Object constraint language, The: Getting Your Models Ready for MDA. Second Edition, Addison Wesley, Boston (2003)
29. Web Services Description Language (WSDL). Version 2.0 (2004) http://www.w3.org/TR/2004
30. Yang, J., Papazoglou, M.P.: Service components for managing the life-cycle of service compositions. Inf. Syst. 29(2) (2004) 97-125
31. Zimmerman, O., Krogdahl, P., Gee, C.: Elements of Service-Oriented Analysis and Design. International Business Machines (2004) http://www-106.ibm.com/developerworks/library/ws-soad1