Communication Papers of the Federated Conference on Computer Science and Information Systems pp. 67–76
DOI: 10.15439/2020F177
ISSN 2300-5963 ACSIS, Vol. 23

Capturing the Evolution of Service-oriented Systems with Architectural Decisions

Szymon Kijas
Warsaw University of Technology
Warsaw 00-661, Poland
Email: szymon.kijas@pw.edu.pl

Andrzej Zalewski
Warsaw University of Technology
Warsaw 00-661, Poland
Email: a.zalewski@elka.pw.edu.pl

***Abstract*—Software evolution is becoming ever more important. SOA is nowadays a well-established and popular software technology. Because of its properties, such as loose-coupling between services and their reconfigurable composition, SOA is an architecture that is particularly suitable for rapidly evolving systems. However, the research on the evolution methodologies for SOA systems is rather scarce. We present a model called MAD4SOA, developed in order to support and capture the evolution of service-oriented systems. Architectural decisions are the first class entities that represent the evolution of a service-oriented system. They are accompanied by a set of relations between model entities and formal integrity constraints. The suitability of the MAD4SOA model has been validated using the real-world example of a system operated in a clearing house company.**

I. Introduction

SERVICE-ORIENTED architectures assume that software systems should be built out of services that are loosely-coupled (easily changeable!) and composable into reconfigurable business processes. This is a great support to systems evolvability. SOA is certainly one of the software technology answers to the rising speed of software evolution. It is now a well-established software technology. Although there are comprehensive methodologies specially crafted for development of SOA systems, RADM [6], Erl's approach [13], SOMA [14], SOMF [1], M. Papazoglou's methodology [15], the research record on the methods supporting the evolution of SOA-based systems is rather scarce. The research presented in this paper is aimed at filling this gap.

We developed a complete methodology, called MAD4SOA, for evolving SOA systems. It comprises: the evolution process, the model for capturing architectural decisions during the evolution, which includes the model of an evolving service-oriented system, set of relations between model entities and integrity constraints. The course of the evolution is captured with architectural decisions, which are first-class entities in the proposed approach. It has been validated on a number of evolution steps of a real-world system operating in a clearing house company. Because of the space limitations, we present here the core concepts of the MAD4SOA methodology, namely, the architecture decision model, set of relations between model entities and its integrity rules. The model has been tailored to the specifics of software evolution – it contains Request for change, which is bound to all the important data describing a single evolution step (requirements, architectural decisions, versions of the SOA system and others). Using a significant fragment of a real-world example, we show that it goes well with the real conditions of service-oriented systems evolution.

The rest of the paper is organised as follows: Section II contains an analysis of related research; Section III presents the entire modelling approach, relations and integrity constraints, as well as the diagrammatical notation; and Section IV sets out a case study that presents the use of the model. Finally, in Section V, we discuss the proposed modelling approach against the related research. The conclusions and research outlook close the paper.

II. Related work

Software architecture is currently perceived as a result of a set of architectural decisions [3], [4], [5]. Architectural decisions can be documented in the form of text records [2], [3], [8], [11] or as diagrammatic models [12], [10]. The model of a single architectural decision usually includes a description of the problem, considered architectural solutions to this problem and their characteristics, an indication of the chosen solution, and the rationale for that choice.

Models of architectural decisions are usually accompanied by a set of relations provisioned for capturing the asscociations between architectural decisions – a number of such sets have been proposed so far [10], [6], [7], [8]. In the existing models, the decisions can also be linked to the engineering artifacts [17] enabling a variety of tracability options.

While the existing research focuses mainly on the modeling of architectural decisions, much less effort has been devoted to the evolution of architectures represented as a set of architectural decisions. The need for capturing the changes to architecture decisions has been indicated in [20]. A special view for architecting and evolving design decisions has been proposed in [16].

Apart from these general-purpose architecture decision models, RADM (Reusable Architectural Decision Model) has been proposed in [6]. It has been specially tailored to assist architecting of SOA systems. It combines architecture decisions with a set of relations, integrity constraints and some inference methods, as well as a means of classification of architectural decisions (levels and topic groups). It is also possible to link the architectural decisions with architectural elements [17].

However, RADM does not explicitly address the evolution of a SOA system. It does not allow for the capture of consecuitive steps of service-oriented architecture evolution.

An attempt to address the evolution of the decisions made during the evolution of service-oriented system has been undertaken in [18], however, the proposed solutions focus only on the decisions, which regard the composition of business processes out of services. SOAD framework [19], in turn, addresses the problem of reusing architectural decisions. There is also an extensive study on the choice between REST and SOAP webservices [21].

Outside the scope of research on architectural decisions, the topic of evolution of SOA systems has been rather scarcely addressed in the existing literature. Existing methodologies of SOA system's devlopment, such as Erl's approach [13], SOMA [14], SOMF [1], M. Papazoglou's methodology [15], address mainly the problem of early evolution of a SOA system (for example: bug fixing at an early stage of operation of the system) as well as aspects of small releases in the course of agile SOA system development [22].

This short survey leads to the conclusion that development of models and methodology supporting the evolution of SOA systems is an open research issue. Its resolution seems to be both important for software engineering practice as currently evolution is the main part of system lifecycle. It can also provide important insights into the intrinsic problems connected with applying the concepts of architecture decision-making in practice. In this paper we focus on the introduction and validation of a model suitable for capturing evolution of SOA systems, while combining a complete development methodology is supposed to be supplemented in later research.

III. Capturing the Evolution of Service-Oriented Systems Architecture

In order to capture the evolution of service-oriented systems architecture, we developed a MAD4SOA model (Maps of Architectural Decisions for Service-Oriented Architecture). MAD4SOA comprises:

- **the model of service-oriented system** – it represents the main entities of a SOA system – section III.A
- **the model of architectural decisions and diagrammatical notation** that will be used in order to document the evolution of a service-oriented system – section III.B;
- **the set of relations** between the model's components (e.g. alternatives, decision problems) – section III.C,
- **the fourteen integrity constraints** (section III.D).

The MAD4SOA model combines a model of a SOA system with a model of architectural decisions. It extends our earlier model and diagrammatical notation named "Maps of Architectural Decisions" (MAD) presented in [10]. Its conceptual roots can also be traced back to RADM model by Zimmerman et al. [6] and Harrison et al. in a paper on architectural decisions [11].

A. Model of a Service-Oriented System

A SOA system model (Fig. 1) comprises three tiers: business processes, services and system components. It should be noted that they follow a scheme similar to the TOGAF [9] standard, though the user interface is not included in our model, as issues related to that are beyond the scope of our research. The model represents the following semantics:

- *Business processes* are composed of the *Activities*, which in turn are achieved by the Services. The latter are indicated by the *realizes* association between the *Activity* and *Service* classes in the UML model.
- The *Services*, in turn, are divided into three categories: 1) *Complex services* 2) *Simple services* and 3) *Foreign services*. A *Complex service* is a composition of other services, a *Simple service* is not a composition of other services and a *Foreign service* is a service delivered by an external provider.
- *Components* implement the functionalities of a *Simple service* and are indicated by the *invokes* association between a *Simple Service* and the *Service Component* classes.
- A *Service Component* may use an *Operational component* in order to implement its functionality. This is represented by the *use* association connecting a *Service Component* with the *Operational Component* class. The use of a database by a service component is a common example of this relation.

Although, the presented model captures the most important entities of the service-oriented system, it can still be expanded, depending on the needs of a given organisa-tion. Let us emphasise that the model does not assume any specific form of docu-mentation of the instances of its entities. This may be just an identifier of a given component in the ITSM register or a set of diagrams, text documents, formal models or even fragments of source code or reference to its repository. The service-oriented system model is used to represent releases of the system that result from the consecu-tive evolution steps – compare section III.

B. Model of Architectural Decisions

The complete model designed to capture the evolution of service-oriented systems with architectural decisions is presented in Fig. 2. The meaning of the entities of the MAD4SOA model of architectural decisions can be explained as follows (italics indi-cate names of classes or associations in the model):

1) *Request for change* (RFC) class represents the typical document prepared in order to launch the modification process. It contains the specification of changes that should be implemented in a given evolution step. The RFC contains the list of architecturally significant *requirements* that must be met by the resolution of a number of *decision problems*. Each solution of such a problem defines the change being either the addition, removal or modification of at least one of the elements

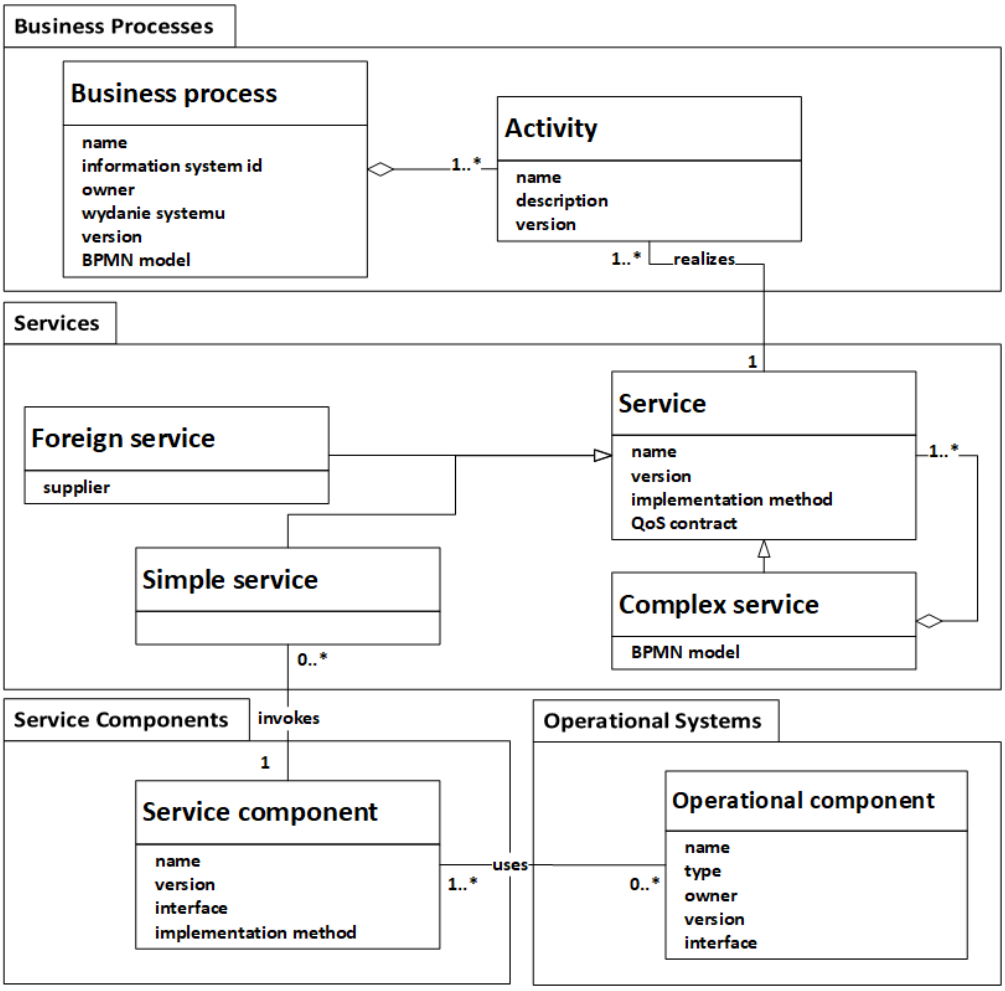

Fig. 1. The model of a service-oriented system.

of an instance of a SOA system model. Resolving all the problems makes a new system release. The current phase of work on the realisation of the *request for change* is represented by one of seven states that the Request for change could be in one of the following states: *defined* - the request for change has been defined and it is waiting for the start of realisation, *evaluated* - the change has been assessed and it is waiting for approval, *approved* - the change has been approved and it is waiting for realisation, *designed* - the change design has been realised, *implemented* - the change has been implemented and deployed, *realised* (after review) - a review of the change has been carried out and the evolution step has been completed, *cancelled* - the change request has been cancelled (it was decided during the acceptance of the change that it will not be implemented).

2) The current release of the system is indicated by the association *old* binding the *Request for change* and *Business process* class.
3) *Requirements* relevant to a given *Decision problem* are distinguished by a *motivates* relation – compare section III.C. The *requirement* can be in one of the three states: *defined* – the requirement has been defined and it is awaiting for realisation, *in progress* – works are ongoing, *completed* - the requirement has been fulfilled (all decision problems related to it have been resolved).
4) *Decision problem* class represents architectural problems that have to be solved. It can be in one of the four states, which represent the stages of the problem's life cycle: *defined* – indicates a newly defined problem, *being solved* – the problem has been created, but it has not been resolved yet, *resolved* – problem has been resolved, *requires reassessment* – indicates that solution, or the occurrence of other problem, requires reconsidering an already resolved problem.
5) There are three types of relations that can link different *decision problems*: *leadsTo*, *constrains* and *decomposesInto* – compare section III.C.
6) The key element of the model of the service-oriented system is the *Business Process*, which identifies the

system and allows navigation to all other elements of the model. The change of any element of the system, e.g. the change of one service, creates a new system that consists of new instances of all elements: *Business process*, *Activities*, *Services*, *Service components* and *Operational components*. Each *Decision problem* relates to a specific, currently considered, instance of the system and is indicated in the model by the input relation. The *Business process* identified by the *input* relation does not have to be identical to the initial process (associated by the *old* relation with the *request for change*). If the decision is made as the n in turn then the previous n-1 decisions may already have modified this process.

7) There can be many alternative Solutions to a given *Decision problem*. Each of these solutions may create a new instance of a given *Business process* and its underlying components (services, service components, operational components). This is reflected by the output relation connecting the *Solution* and *Business process* classes. The analysis of a given solution can be in one of the four states: *defined* – assigned immediately after creating an element; *feasible* – indicates a solution meeting all the requirements, *infeasible* – indicates a solution that does not meet at least one of the requirements, *chosen* – indicates the finally selected solution.
8) Each of the alternative *Solutions* is supposed to be assessed in terms of its *Pros* and *Cons*. They are described by: a textual *description*; *significance*: low, medium or high; *related requirements*. The latter is optional and indicates: in case of cons – the requirements that cannot be met because of a given cons; in the case of pros – the requirements whose fulfilment is guaranteed by the given pros.
9) *Decision maker* class represents the architects that resolve a *Decision problem*.

In practice, the MAD4SOA models should be created with the use of a diagrammatical notation summarised in Fig. 3. It is an extension of our previous MAD notation, made in order to capture the additional elements necessary to document the evolution of SOA systems.

Similarly to MAD, MAD4SOA notation comprises two types of diagrams: an Architecture Decision Relation Diagram (ADRD) and an Architecture Decision Problem Map (ADPM). ADRD represents the identified decision problems, while ADPM models an individual decision problem. Despite obvious similarities, there are many extensions made to MAD diagrams, aimed at including all the elements of the MAD4SOA model. The ADRD diagram now includes symbols that represent requests for change, requirements and business processes. As well as solutions of decision problems that produce a new release of the system. The ADPM diagram can now contain more than one decision problem. This is necessary in cases of a problem decomposition, as well as if a relation between the solutions to various problems has to be shown.

C. Relations

The MAD4SOA model comprises a set of relations that may be used in order to capture the real-world relations that may exist between: decision problems, decision problems and requirements, decision problems and solutions of other problems, as well as between the solutions of different decision problems. For the purpose of the definitions given below, let: R – denote a set of requirements, P – a set of architectural problems, and A – a set of considered solutions (alternatives).

The motivates relation:

$$motivates \subseteq R \times P$$

It connects the requirements with the decision problems that have to be solved in order to meet the requirements.

For example: take a requirement to implement a new functionality of verifying a personal ID number. To fulfil this requirement, two decision problems have to be solved: “Which service should be used in order to verify the personal ID number?" and “In which step of the business process does this service have to be invoked?”.

The leadsTo relation:

$$leadsTo \subseteq P \times P$$

In many cases, one decision problem implies the need to resolve another problem. Such problems should be connected with leadsTo relation.

For example: The problem of “choosing webservice implementation technology” makes it necessary to resolve a problem concerning the choice of the runtime environment for our webservices (it could be one of many application servers available on the market, or the application server already used by the corporate systems). Therefore the problems “Which type of webservices?” and “Which runtime environment?” are connected by the leadsTo relation.

Properties: The leadsTo relation is irreflexive, antisymmetrical and transitive. Therefore, this is a relation of acute partial order in the set of problems P.

The decomposesInto relation:

$$decomposesInto \subseteq P \times P$$

It captures the decomposition of a given decision problem into several sub-problems. In practice, there are often cases, in which in order to solve a complicated decision problem, it is necessary to decompose it into several smaller sub-problems. The decomposed problem can be assumed as resolved only if all its sub-problems have been resolved.

For example: the problem of “choosing the software implementation technology” may comprise two sub-problems, namely: “choice of a programming language?” and “Choice of a mechanism of communication with the database?”.

Properties: The decomposesInto relation is reflexive, antisymmetrical and transitive. Therefore, this is a relation of acute partial order in the set of problems P. There must be no cycles in the graph of this relation.

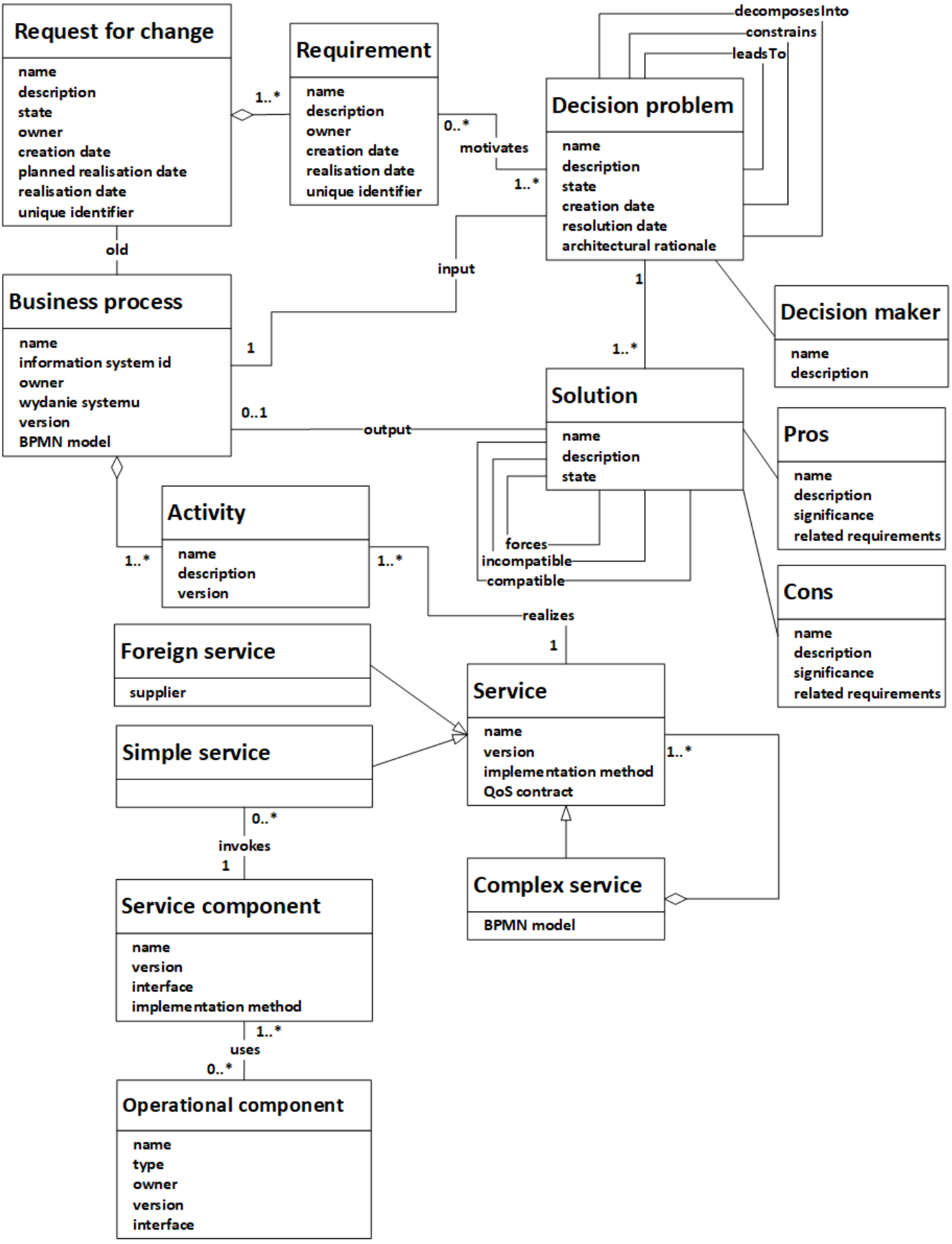

Fig. 2. Model of architectural decisions in the evolution of a service-oriented system.

The constrains relation:

$$constrains \subseteq P \times P$$

There are often situations in which the solution of a given decision problem limits the scope of possible solutions to another problem. This is captured by the constrains relation.

For example: take two decision problems: "the problem of database selection" and the problem of "choosing the mechanism of data replication from the selected database to the MS SQL Server 2014 database". The choice of the Oracle 12c database will exclude the use of the replication mechanism provided with the MS SQL Server 2014 database, because MS SQL Server 2014 does not support replication from Oracle version 12c. Therefore, there is a constrains relation between these problems.

Properties: The constrains relation is irreflexive and transitive. There must be no cycles in the graph of this relation.

The compatible, incompatible and forces relations:

$$compatible \subseteq A \times A$$

$$incompatible \subseteq A \times A$$

$$forces \subseteq A \times A$$

In many cases, some solutions of different problems can exist together (they are compatible) or they cannot (they are incompatible) in the system architecture. Sometimes the relation between solutions is even stronger, so that choosing a certain solution simply forces us to choose a specific solution to another problem.

For example: take the problems of choosing the system implementation technology and selecting a specific application server that will be the execution environment for the designed software.

In the case of choosing J2EE as the implementation technology, we can choose both Jboss and Weblogic as the application server, because both of them support J2EE technology. There is a compatible relation between the J2EE and the Jboss solutions and Weblogic solution.

In the case of choosing J2EE technology, we cannot choose the IIS application server, because IIS does not support J2EE technology. There is an incompatible relation between the J2EE and IIS solutions.

In the case of choosing .NET as the implementation technology, we then have to choose IIS as the application server, because only this application server supports .NET technology. Therefore there is the forces relation between the .NET solution and the IIS solution.

Properties: The compatible relation is reflexive and symmetric. The incompatible relation is irreflexive and symmetric. The forces relation is irreflexive and antisymmetric.

D. Integrity constrains of the model

The relations between the elements of the MAD4SOA model of architectural decisions are not independent of each other. Some relations between decisions can exist at the same time, while others cannot. The integrity constrains make it possible to discover potential flaws in an instance of the MAD4SOA model crafted by an architect. Fourteen integrity constrains for the MAD4SOA model are presented below.

Integrity constraint 1: The decomposesInto and the leadsTo relations are mutually exclusive. The decomposed decision problem can only be solved when all subsequent problems raised as a result of its decomposition have been resolved. This would be superfluous to treating the problems resulting from the decomposition of a given problem as being enforced by the decomposed problem.

Integrity constraint 2: The decomposesInto and the constrains relations are mutually exclusive. The decomposed decision problem can only be solved if all its sub-problems are resolved. The solution of the decomposed problem is the superposition of the solutions to the sub-problems. Therefore, the solution to the decomposed problem cannot constraint the scope of the solutions to sub-problems.

Integrity constraint 3: If there is a constrains relation between two decision problems, then there is at least one pair of their solutions that are incompatible together (they are in an incompatible relation). The above property results from the meaning of the constrains relation. If problem A limits the set of solutions available to problem B, it means that there must be solutions to problem B that are incompatible with one or more solutions to problem A.

Integrity constraint 4: The compatible and the incompatible relations are mutually exclusive. It is impossible for two solutions to be both compatible and incompatible at the same time.

Integrity constraint 5:

$$forces \subseteq compatible$$

If a certain solution to one problem forces the adoption of a specific solution of another problem in the same system, then the consistency of the system design requires that both solutions can coexist with each other.

Integrity constraint 6: The forces and the incompatible relations are separable. Because the forces relation is a subset of the compatible relation, it means that the forces and the incompatible relations are also mutually exclusive.

Integrity constraint 7: The forces relation cannot occur between solutions of the same decision problem. The next rules define the meaning of the states of the individual model elements.

Integrity constraint 8: The requirement may change its state to completed only if all related decision problems are in the resolved state. The requirement is considered as met only if it is possible to resolve all the problems connected with this requirement. That means that all decision problems arising as a result of this requirement will be resolved.

Integrity constraint 9: The RFC can go into the completed state only if all the requirements associated with it are in the completed state. The request for change can only be considered complete once all its architecturally significant requirements have been met.

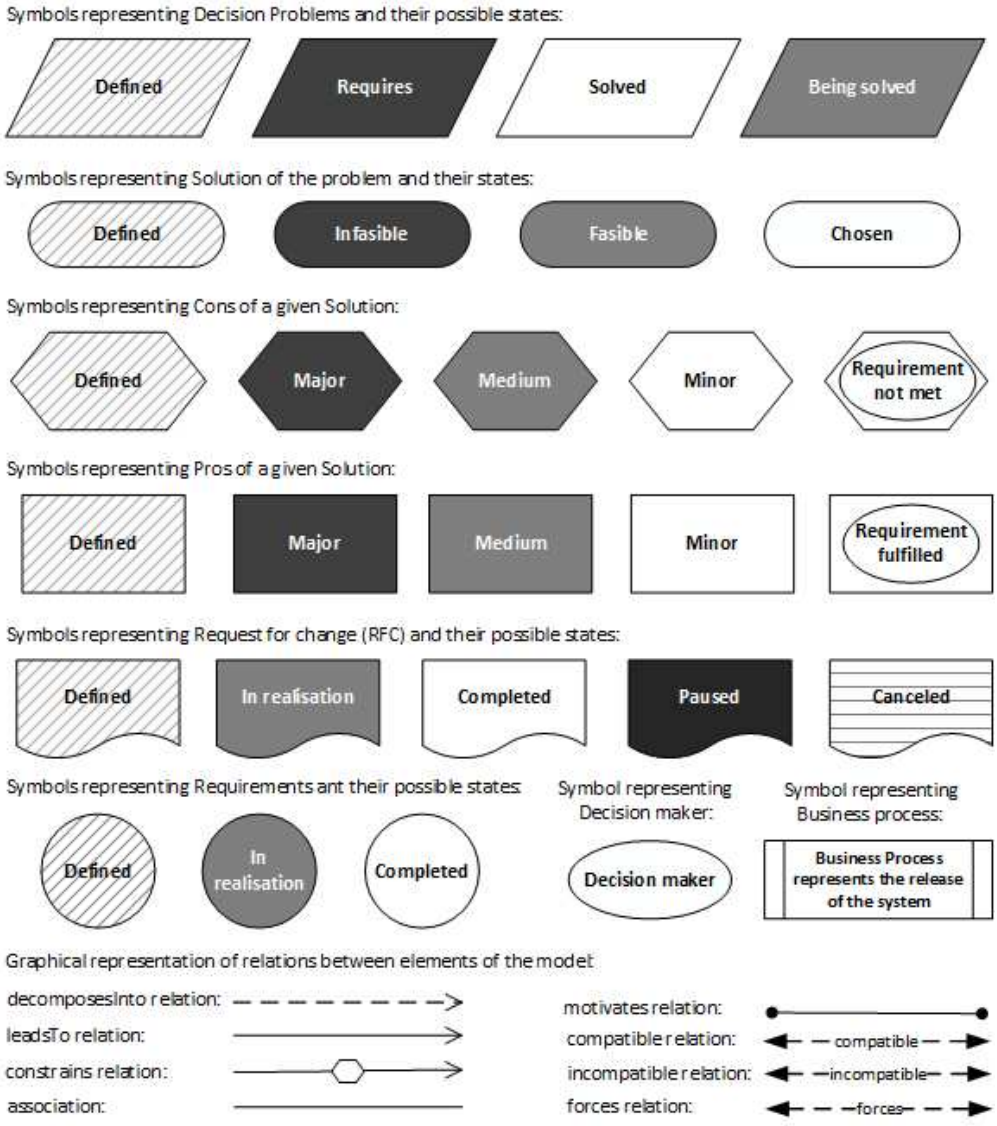

Fig. 3. The notation representing components of the MAD4SOA model.

Integrity constraint 10: At most, one solution of a decision problem can assume the chosen state. The system design must be unambiguous. That means that for each considered decision problem there must be only one clearly indicated chosen solution.

Integrity constraint 11: The decision problem may go into the resolved state only if one of its solutions assumes the chosen state.

Integrity constraint 12: If at least one cons of a solution goes into the requirement not met state, then this solution cannot be selected as the resolution of the decision problem, and should therefore go into the infeasible state.

Integrity constraint 13: There must be no incompatible relation between any two solutions in the chosen state (these solutions concern different problems). All resolutions of decision problems must coexist with each other in the same system to ensure system design consistency.

Integrity constraint 14: If there is a forces relation between the two solutions, then if one of them is chosen, the other must also be selected. Some solutions of decision problems cannot exist independently in the system.

IV. The Case Study – Validation of MAD4SOA Model

This section presents a case study illustrating the use of the MAD4SOA model for capturing the evolution of a real service-oriented system. This case study is based on the clearing system operating in the Polish banking system. The system has been designed to settle instant transfers carried out between two banks. The clearing system verifies each payment order, confirms the consent of both banks to carry out the transaction and registers the change in the account balances

of both banks and in the Central Bank system. Performing these activities requires the interaction of the clearing system with the systems of banks participating in a transaction (the ordering bank and the bank receiving the transfer) and with the system of the Central Bank. The evolution process of the system has produced six releases of the system in several evolution steps:

- Creation of functionality of instant payments between commercial banks as the first release
- Adding instant payments to the Central Bank (tax payments) as the second release
- Adding a simple functionality of complaints as the third release
- Adding a Back Office module as the fourth release
- Adding postponed payments as the fifth release
- Removing one type of tax payment as the sixth release.

The lifecycle of the system has not been finished yet.

As a part of the example, the evolution of the Back Office module of the clearing system has been presented. This module enables customer complaint handling, as well as the parameterisation and monitoring of the system. These activities were carried out manually by making changes to the tables in the database using previously prepared scripts in earlier stages of the life cycle of the system. Next, the Back Office has been implemented to enable the automation of these activities.

The complaint handling process is the most important functionality of the Back Office application. Complaints are generated if the bank ordering the transfer does not have sufficient funds on the internal account, or the bank receiving the transfer does not confirm the transfer. If necessary, the administrator can make a decision and manually modify the transaction state.

Several decision problems had to be resolved by the architect during the design of the Back Office functionality during both evolution steps. The ADRD diagram that presents the decision problems solved by the architect is presented in Figure 4. The example of the ADPM diagram presented in Figure 5 describes how the problems identified during this evolution step (step 4) have been resolved. The most important is the evolution of the decision problem: *"How to implement the complaint handling functionality?"*. This problem had been resolved in the current version as *"Using predefined scripts"* during the previous evolution step (step 3). It had to be revised due to implementing a new RFC during the next evolution step (step 4). Both solutions to this problem produced new releases of the system in subse-quent evolution steps. It is represented by output relation linking solutions and in-stances of business processes.

The business process representing the third release of the system and a Request for Change describing changes implemented in the fourth evolution step are connected by the old relation. It means that this RFC applies to the modification of the third release of the system. The business process representing the third release of the system is also linked with the *"How to implement the complaint handling functionality?"* decision problem resolved in the fourth evolution step by the input relation. This means that the third release of the system has been modified and transformed to the fourth release. Additionally, this ADRD diagram represents several decision problems that have been resolved during the fourth evolution step. The detailed process of solving those problems is presented in Figure 5.

V. Discussion

The MAD4SOA model, which is the core contribution of this paper, has been tailored to the needs of documenting and supporting the engineering in the course of evolution of service-oriented systems. It provides for linking decisions made during different evolution steps, which makes it possible to trace the sequence of actions taken by architects during the development of subsequent releases of the system. An important feature of the proposed model is the inclusion of request for change document and its requirements as a model's entity. This brings the model closer to the real-world practice. All these unique features of MAD4SOA model constitute the core of our contribution.

Naturally, there are many similarities between proposed models and alternative architecture decision models presented in Section II, which is inevitable. They concern mainly:

- the model of architectural decisions, which can be traced back to our earlier research on MAD model [10] as well as to classical works on architecture decisions [5];
- some relations (e.g. decomposesInto) and integrity constraints (e.g. integrity constraint No. 6) can be easily traced back to the RADM model [6];
- tracability machanism linking diagramatical architectural models (e.g. in BPMN) with architectural decisions can also be found in [17].

An added value that MAD4SOA delivers is also an intuitive diagrammatic notation for representing architectural decisions, which is similar to popular mind maps. In contrast to RADM, we do not introduce any scheme or mechanism for classifying architectural decisions similar to topic groups or levels. Our internal experiments, have shown that this in too many cases leads rather to confusion than to clarity. In our model all the changes are linked to business processes, which they may finally affect.

The main limitation of the proposed approach results from the complexity of the architecture-decision model and its associations with the models of SOA system (e.g. business process models in BPMN), which may hinder its use. Naturally, more extensive evaluation and tool support is necessary.

VI. Conclusion

The complete model supporting the capturing of architectural decisions during the evolution of a service-oriented system has been proposed. It combines an architectural decision model and a model of a service-oriented system. Thanks to this, MAD4SOA supports capturing architectural knowledge created during the evolution of a service-oriented system. The use of the model has been validated on the evolu-tion of a

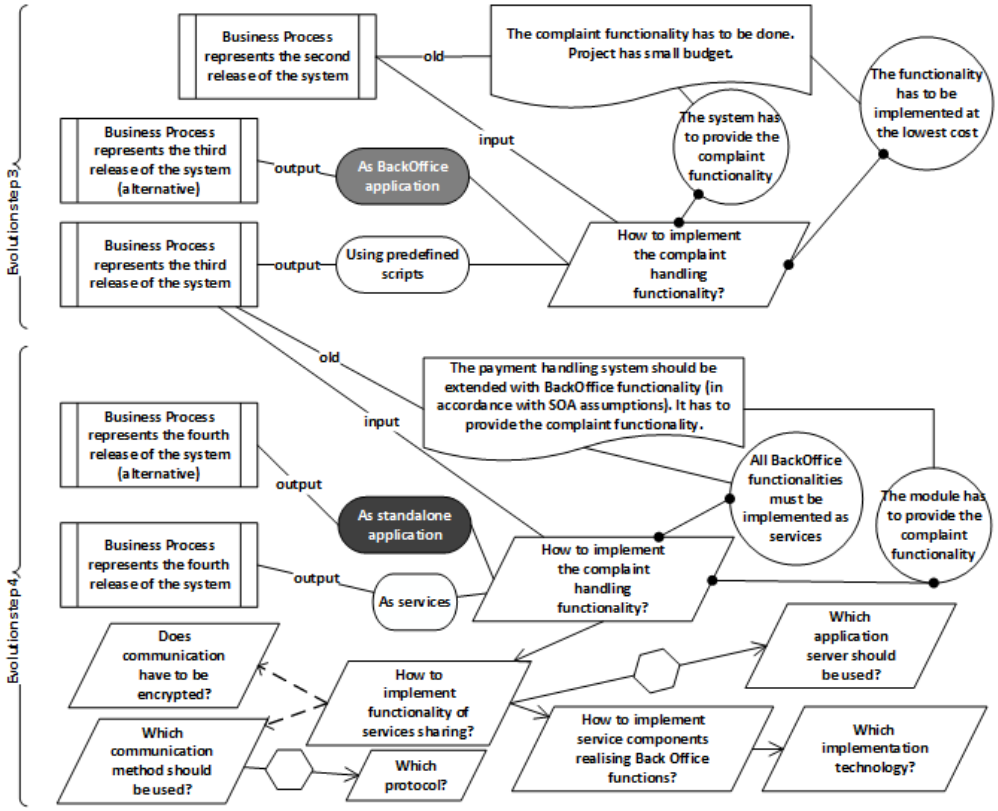

Fig. 4. The ADRD diagram presenting problems resolved during the design of the new release.

real service-oriented system operating in the Polish banking system.

Further work shall include defining formal semantics of MAD4SOA model and developing supporting tools.

REFERENCES

[1] M. Bell, "SOA Modeling Patterns for Service-oriented Discovery and Analysis," *Wiley and Sons,* 2010.

[2] M. Ali Babar, et al.,"Architecture knowledge management. Theory and Practice," *Springer – Verlag Berlin Heidelberg,* 2009.

[3] J. Tyree, A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software,* vol. 22, iss. 2, 2005, pp. 19–27.

[4] ISO/IEC, "ISO/IEC/IEEE 42010:2011: Systems and software engineering – Architecture description," *ISO/IEC,* 2011.

[5] A. Jansen, J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," *5thWorking IEEE/IFIP Conference on Software Architecture,* 2005, pp. 19–27.

[6] O. Zimmermann, et al., "Managing architectural decision models with dependency relations, integrity constraints, and production rules," *Journal of Systems and Software,* vol. 82, no. 8, 2009, pp. 1249–1267.

[7] P. Kruchten, P. Lago, H. van Vliet, "Building up and reasoning about architectural knowledge," *QoSA,* 2006.

[8] P. Kruchten, "An Ontology of Architectural Design Decisions," *Proc. of 2nd Groningen Workshop on Software Variability Management,* 2004, pp. 54–61.

[9] The Open Group, "The Open Group Architecture Framework (TOGAF®) Version 9.1," *The Open Group,* http://pubs.opengroup.org/architecture/togaf9-doc/arch/ 2018.

[10] A. Zalewski, S. Kijas, D. Sokołowska, "Capturing Architecture Evolution with Maps of Architectural Decisions 2.0," *ECSA 2011 Lecture Notes in Computer Science,* vol. 6903, 2011, pp. 83–96.

[11] N. B. Harrison, P. Avgeriou, U. Zdun, "Using Patterns to Capture Architectural Decisions," *IEEE Software,* vol. 24, no. 4, 2007, pp. 38–45.

[12] M. Shahina, P. Lianga, M. Ali Babar, "A systematic review of software architecture visualization techniques," *The Journal of Systems and Software,* vol. 94, 2014, pp. 161–185.

[13] T. Erl, "Service-Oriented Architecture: Concepts, Technology, and Design, Upper Saddle River," *Prentice Hall PTR,* 2015.

[14] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, K. Holley, "SOMA: A method for developing service – oriented solutions," *IBM Systems Journal,* vol. 47, no. 3, 2008, pp. 377–396.

[15] M. P. Papazoglou, W. J. van den Heuvel, "Service-oriented design and development methodology, International Journal of Web Engineering and Technology," *IJWET,* 2006.

[16] R. Capilla, F. Nava, J. C. Dueñas, "Modeling and Documenting the Evolution of Architectural Design Decisions," *Proc. 2nd Workshop Sharing and Reusing Architectural Knowledge Architecture,* 2007.

[17] R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou, J. M. Küster, "An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle," *ECSA 2011 Lecture Notes in Computer Science,* vol. 6903, 2011, pp. 303–318.

[18] O. Zimmermann, J. Grundler, S. Tai, F. Leymann, "Architectural Decisions and Patterns for Transactional Workflows in SOA," *ICSOC 2007 Lecture Notes in Computer Science,* vol. 4749, 2007, pp. 81–93.

[19] O. Zimmermann, "Architectural Decisions as Reusable Design Assets," *IEEE Software,* vol. 28, 2011, pp. 64–69.

[20] M. Nowak, C. Pautasso, O. Zimmermann, "Architectural decision modeling with reuse: challenges and opportunities," *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge,* 2010, pp. 13–20.

[21] C. Pautasso, O. Zimmermann, F. Leymann, "Restful web services vs. "big"' web services: making the right architectural decision," *Proceedings of the 17th international conference on World Wide Web,* 2008, pp. 805–814.

[22] X. Wang, N. Ali, I. Ramos, R. Vidgen, "Agile and Lean Service-Oriented Development: Foundations, Theory, and Practice," *IGI Global,* 2012.

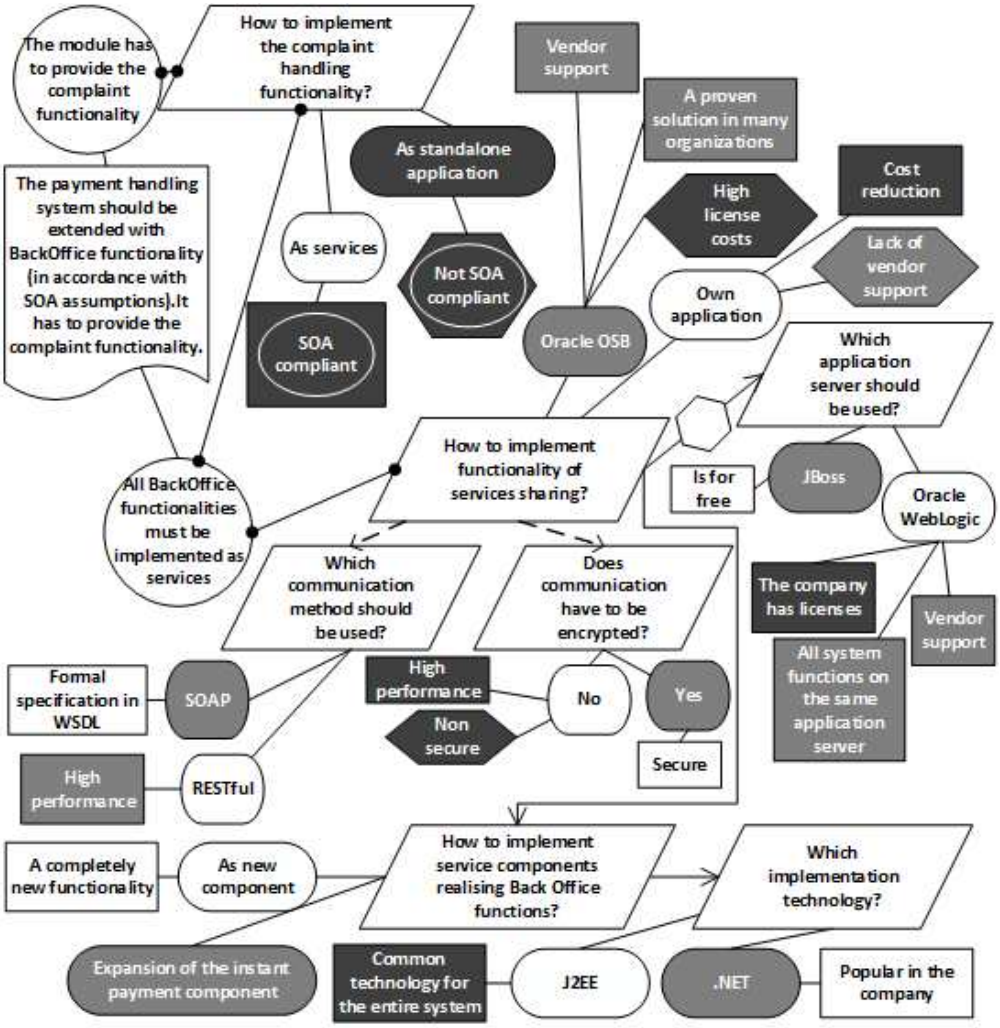

Fig. 5. The ADPM diagram presenting how to be resolve the problems arising during the design of the new system release.