Journal of King Saud University – Computer and Information Sciences (2014) **26**, 296–307

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Managing OAM&P requirement conflicts

Zohair Chentouf

College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia

Received 24 October 2013; revised 5 December 2013; accepted 13 March 2014
Available online 9 May 2014

KEYWORDS

Requirement engineering;
KAOS;
Service management;
Service design

Abstract Specifying consistent services at early project stages is a telecommunication service engineering challenge. Service logic inconsistencies, also known as feature interactions (FIs), can affect various types of services ranging from signaling protocol features to value-added end user services. This problem has been investigated for all those types of services. However, inconsistencies of Operation, Administration, Management and Provisioning (OAM&P) services have not been sufficiently addressed. The present paper studies the detection of OAM&P service inconsistencies at the software requirement specification stage. The aim is at the resolution of the problem before reaching the implementation step. The basic idea of the here reported approach is to consider service inconsistencies as software requirement conflicts. The contribution of the present paper consists of an OAM&P requirement modeling language and a requirement conflict detection method. A validation with a case study is reported.

1. Introduction

The feature interaction (FI) problem arises when two or more services, running together, interact in such a way that at least one behaves in an undesirable manner. In emerging telecommunication architectures built around packet-switched networks, the FI is even more complex due to the characteristics of these architectures. In fact, the convergence of a variety of networks implies service complexity, application distribution, technology heterogeneity, multi-vendor involvement, and user programmability. All these aspects render the FI problem management a more difficult endeavor.

The FI problem has been studied in circuit-switched networks (Amyot and Logrippo, 2003; Calder and Magill, 2000; Dini et al., 1997; Du Bousquet and Richier, 2007; Kimbler and Bouma, 1998; Nakamura and Reiff-Marganiec, 2009; Reiff-Marganiec and Ryan, 2005), but there are still no completely satisfactory solutions. Most of the FI detection research uses formal methods. Services are modeled using a formal language. Then, formal techniques are used to detect possible interactions. The commonly used formal techniques are temporal logic (Blom, 1997), theorem proving (Gammelgaard and Kristensen, 1994), Petri nets (Nakamura et al., 1997), extended finite state automata (SDL language, for example) (Gibson and Mery, 1997), and process algebra (LOTOS language, for example) (Amyot et al., 2000). Informal methods are also used to detect FI. For example, Charnois (1997) uses natural language processing to identify interactions between textual representations of service logic. Cherkaoui and Khoumsi (2002) proposed a solution based on software agents. Kolberg and Magill (2001) and Amer et al. (2000) designed negotiating agents that try to satisfy the preferences of end

E-mail address: zchentouf@ksu.edu.sa

Peer review under responsibility of King Saud University.

ELSEVIER Production and hosting by Elsevier

1319-1578 © 2014 King Saud University. Production and hosting by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jksuci.2014.03.004

users and network operators. A complete survey of the literature can be found in Calder et al., 2003. In packet-switched networks, FI started to be studied a few years ago (Chentouf and Khoumsi, 2013; Chentouf et al., 2003a,b,c; Lennox and Schulzrinne, 2000; Rizzo and Garyfalos, 2000).

In this article, we are interested in studying the FI problem in OAM&P systems. OAM&P service design has been widely studied. For example, in McKiou and Buckley, 2003, the authors designed a UTRAN object-oriented OAM&P system. In Silverman et al., 2000, the problem of rapid and easier service creation and service and network management was investigated. In Donadio et al., 2009, a service-oriented OAM&P framework based on TMN, Web services and software agents was proposed. In Lavinal et al., 2009, a self-adaptive multi-agent-based OAM&P framework was designed. The authors of Charcranoon et al., 2005 elaborated an OAM&P architecture for Optical Burst Switched (OBS) networks. In Modarresi and Mohan, 2000, the authors discussed the challenges and opportunities associated with the unified control and management of next-generation networks. However, in all these research works, the FI problem has not been studied.

Compared with the number of FI research works in communication services, the works in OAM&P FI remain very few. Chi et al. (2003) analyzed the problem of interactions that can occur between call control services and management protocol features of optical networks. The authors proposed some general guidelines to detect and resolve these problems. Ilić et al. (2006) reported on the enhancement of a system that allows service designers to specify high-level inter-operation services between multiple communication networks, including management features. Georgatsos et al. (1997) claimed an important role of the Telecommunications Management Network (TMN) functional layers in handling FI in the PSTN. The authors provided a set of recommendations about the function of some of the TMN services. The latter three research works did not produce a FI detection and resolution procedure. We believe that the FI research community has not been interested in the problem of OAM&P FI because the focus has been on communication and communication control services. To our best knowledge, FI specifically in the OAM&P layer has only been studied by Chentouf (2012). There are many advantages of the present article compared with Chentouf, 2012. First, OAM&P FI is addressed at the requirement software stage. Detecting OAM&P requirement conflicts allows engineers to solve and effectively prevent inconsistencies from propagating along the subsequent stages of the software process. In the sequel, we will use "requirement conflicts" to note "feature interactions seen as requirement conflicts". A second advantage of the present work over Chentouf, 2012 consists of deriving an OAM&P requirement modeling language from a standard and well-known requirement modeling language called KAOS. As a direct consequence, more types of FI can be detected. Another advantage of the present work is that the consistency and computational complexity of the proposed solution have been studied. A more elaborate comparison between the two research works will be presented in Section 5.

The requirement conflict detection method proposed in the present work is the same as that for FI: modeling items (requirements), then comparing pairs of item models to detect possible interactions. The solution here proposed assumes that most requirement engineers still use natural language to write requirements. Indeed, according to a field survey (Mariangela and Pierluigi, 2004), 71.8% of requirement documents are written in natural language. This fact motivated us to base our requirement modeling on natural language. Therefore, we propose a controlled natural language structure in terms of an Extended Backus-Naur Form (EBNF) to be the basis of a requirement writing automatic tool. Thus, requirement analysts are assumed to write requirements using such an automatic tool instead of an ordinary text editor. For the proposed EBNF to be able to capture the semantics of OAM&P requirements, it has been elaborated as an interpretation of KAOS (Objectiver, 2007), the well-known requirement modeling language. A requirement conflict detection method has been based on the EBNF. The consistency, completeness, and computation complexity of the proposed solution have also been studied. The research result validation has been performed through a set of proof-of-concept examples.

The remainder of the paper proceeds as follows. We begin by introducing requirement modeling research in Section 2. We outline our OAM&P requirement modeling language in Section 3. We describe the proposed OAM&P feature interaction detection method in Section 4, including a study of the method's consistency, completeness, and computational complexity. Section 5 examines related works. Validation of the method is described in Section 6 through a case study. Section 7 concludes the paper.

2. Modeling requirements

Requirement modeling languages (RML) typically encompass concept and relation modeling methods. Some RML contain automated procedures to implement search queries (Jureta et al., 2010). Early-phase RML includes RML (Greenspan et al., 1986) and ERAE (Dubois et al., 1988). The ontology in the latter language was judged to be limited (Greenspan et al., 1994). KAOS (Dardenne et al., 1993) and i* (Yu, 1997) have richer ontology, central to which are the concepts of system and stakeholder goals. Van Lamsweerde (2000a) defines a goal as "a prescriptive statement of intent that the system should satisfy through cooperation of its agents". Telos (Mylopoulos et al., 1990) adopted a different approach: putting in the language itself the facilities required to build the ontology.

In this paper, we are interested in KAOS, which is a goal-directed requirement engineering methodology, not simply a modeling language (Dardenne et al., 1993; Darimont and Van Lamsweerde, 1996; Van Lamsweerde, 2001; Van Lamsweerde and Willemet, 1998b; Van Lamsweerde et al., 1995). The choice of KAOS was motivated by the fact that a KAOS environment is available and has been used in large-scale industrial projects (Darimont et al., 1998). The KAOS method consists of (i) eliciting and decomposing goals, (ii) deriving objects and operations from goals, and (iii) and eliciting requirements on the objects and operations to meet the goals. The underlying ontology includes a number of concepts: object, operation, agent, and goal. It also contains relations: performance, aggregation, composition, and inheritance (Dardenne et al., 1993; Objectiver, 2007; Van Lamsweerde and Letier, 2000b; Van Lamsweerde et al., 1998a). KAOS allows analysts to build a glossary progressively and simultaneously during requirement definition. The glossary consists of UML class diagrams where the system-to-build's classes and their attributes and relations are elicited (Objectiver, 2007).

The KAOS language combines two parts. The first one consists of semantic nets (Brachman and Levesque, 1985) for the conceptual modeling of goals, requirements, agents, objects and operations of the system-to-be. The second component is optional. It consists of a temporal logic (Koymans, 1992) for the specification of goals, requirements, and objects. The analyst can use this component to formalize the conceptual modeling part into a temporal first order logic theory. However, requirement analysts and stakeholders are often not familiar with formal logic (Gervasi and Zowghi, 2005), which is why this module is not expected to be used at the beginning of the requirement process. Its aim is verifying the satisfaction of requirements (i.e., checking if properties satisfy goals) after they have been written (Objectiver, 2007). Fig. 1 depicts an example written in the graphical language of KAOS. It summarizes some requirements of an elevator system: "When a passenger pushes a button of the elevator system, the system refreshes the list of instructions (Reschedule) that the elevator controller has to execute. The new schedule will be immediately in use by the elevator controller" (Objectiver, 2007). Fig. 1 shows that "Passenger command" is an event, "Reschedule" is an operation, and "Schedule" is an object.

3. Modeling OAM&P requirements

To detect OAM&P requirement conflicts, they have to be modeled in a suitable language. For this aim, we propose a modeling language derived from KAOS. In this section, we introduce the constructs of this language. We will explain in Section 4 how this language can be used by the requirement analysts to specify requirements and detect conflicts among them.

Definition 3.1 (*Object*) An object is anything of interest that is subject to an operation (action).

The set of objects will be denoted Obj. Inheritance (is-a relation), aggregation (has-a relation), and composition (contains-a relation) are supported.

Definition 3.2 (*Object path*) An object path consists of m objects $(o_1, \ldots, o_m)$, $m \geqslant 2$. It notes the existence of a relation between every pair of objects (o_i, o_{i+1}). A relation can be any one of those mentioned in Definition 3.1.

The set of paths is U_n Obj^n, $n = 2, |Obj|$. The relation between (o_i, o_{i+1}) may be different from the one between (o_j, o_{j+1}), $i \neq j$.

Definition 3.3 (*Agent*) Any processor of some operation on some object(s). Agents include humans, devices, programs, etc.

The set of agents will be denoted Agt.

Definition 3.4 (*Operation*) An operation is performed by an agent. It has inputs and outputs that are objects.

Let Opr be the set of operations. Operations will be written as in KAOS, i.e., they contain both the verb and the object. Example: access-account (Objectiver, 2007).

Definition 3.5 (*Event*) An event is the pre-condition of an operation. It may either start (cause) or stop the requirement's operation (action). It may also forbid an operation.

The set of all events will be denoted Evt. As recommended by KAOS (Objectiver, 2007), events are written in a passive form and are considered as a single term. For example, credit limit reached will be written as follows: credit-limit-reached.

Definition 3.6 (*ID*) A natural number that exhaustively identifies a requirement.

Definition 3.7 (*Alphabet*)

The OAM&P modeling language comprises the following:

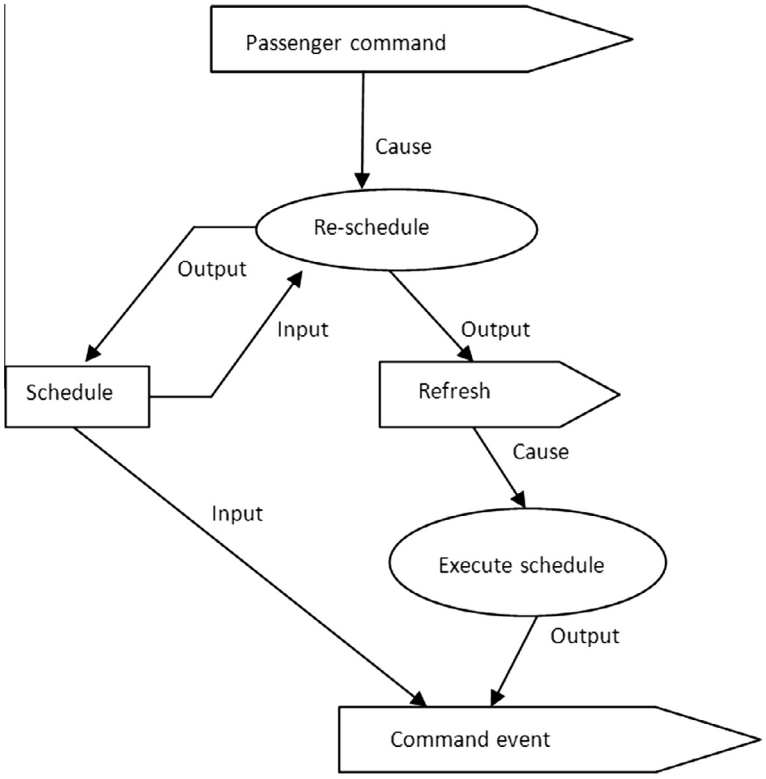

Figure 1 KAOS requirement modeling example.

- Constant symbols: Every member of Obj U Agt U Opr U Evt U N U {START, STOP, FORBID, VOID, ALL} U_n Obj^n, where N denotes the set of natural numbers.
- Variable symbols: We define nine sets, V_{obj}, $V_{obj\text{-}set}$, V_{agt}, V_{opr}, V_{evt}, V_{ids}, V_{hld}, V_s, and V_{pth}, of variable symbols ranging over the sets Obj U {VOID}, non-empty subsets of Obj, Agt U {ALL}, Opr, Evt U {VOID} U N, N, N U {VOID}, {START, STOP, FORBID}, and U_n Obj^n, respectively.

V_{obj} = {object, $object_1$, $object_2$, ... }
$V_{obj\text{-}set}$ = {object-set, $object\text{-}set_1$, $object\text{-}set_2$, ... }
V_{agt} = {agent, $agent_1$, $agent_2$, ...}
V_{opr} = {operation, operation1, $operation_2$, ...}
V_{evt} = {event, $event_1$, $event_2$, ...}
V_{ids} = {id, id_1, id_2, ...}
V_{hld} = {hold, $hold_1$, $hold_2$, ...}
V_s = {s, s_1, s_2, ...}
V_{pth} = {path, $path_1$, $path_2$, ...}

- Predicate Symbols: We consider the following predicate symbols:

a. A binary predicate symbol, *interaction*. The arguments of *interaction* are requirement identifiers. If *interaction*(id_1, id_2) is true, then the two requirements identified by id_1 and id_2 are in conflict.

b. A binary predicate *reachable* whose arguments are objects. If *reachable*($object_i$, $object_j$) is true, then there exists a path of m objects ($object_1$, ..., $object_i$, ..., $object_j$, ..., $object_m$), $m \geqslant 2$ (see Definition 3.2).

c. A unary predicate *error*. Its argument is a requirement identifier. The predicate holds if the requirement referenced by the identifier is not well-formed.

d. Three binary predicate symbols, *c*, *g*, and *h*, with objects as arguments. *c*($object_1$, $object_2$), *g*($object_1$, $object_2$), and *h*($object_1$, $object_2$) are true if $object_1$ contains (composition), has (aggregation), or specializes (inheritance) $object_2$, respectively.

- Logical operators: The operators NOT (negation), AND (conjunction), and OR (disjunction).

Definition 3.8 (Requirement tuple) A requirement is an 8-tuple (id, s, event, agent, operation, object-set, object, and hold), where id, s, event, agent, operation, object-set, object, and hold are variables that belong to the variable sets V_{ids}, V_s, V_{evt}, V_{agt}, V_{opr}, $V_{obj\text{-}set}$, V_{obj}, and V_{hld}, respectively.
The semantics of a requirement tuple is the following:

- id: the identifier of the requirement. We chose to represent id as a natural number.
- agent: the agent who performs the operation. If the action shall be executed by all the agents, then agent will be set to the constant ALL.
- operation: the operation to be performed by the agent.
- *s*: the effect of the requirement start, stop, or forbid an operation. START means that the event gives the agent allowance or obligation to perform the operation. STOP means that the event causes the operation to stop. In such a case, the operation designates a continuous or iterative process or service. FORBID forbids the agent to execute the operation. In the sequel, we will call the values of *s* *s*-values.
- event: the event that starts, stops, or forbids the requirement's operation, depending on the content of *s*. The event may contain a date and time. In this case, START, STOP, or FORBID will be executed at the specified date and time. We chose to represent the date and time as the number of milliseconds since 1970/01/01. This representation is adopted by many programing languages, including Java, for example. If the requirement's operation is not conditioned by an event, event will be set to the constant VOID. In such a case, s may be set to either START, to mean unconditional allowance or obligation, or FORBID for unconditional forbiddance.
- object-set: a set of objects that constitutes the input of the operation.
- object: the object that is the output of the operation. Every requirement has only one output. The aim is to ease the requirement interaction detection. If an operation has more than one output, it shall be split into more than one requirement tuple so that every tuple has only one output object.
.- hold: this value is used if the requirement contains an operation that has to be executed periodically. In this case, hold shall contain the number of milliseconds to wait before re-executing the operation. If there is no specification of repetition for the operation, hold will be set to the constant VOID.

Here are some requirement tuple examples:
Example 3.1: all agents shall not delete invoices.
(23, FORBID, VOID, ALL, delete-invoice, invoice, invoice, VOID)
Example 3.2: once a subscriber has made the first payment, she shall be able to access her account.
(566, START, subscriber-first-payment-done, subscriber, access-subscriber-account, subscriber-account, subscriber-account, VOID)
Example 3.3: on May 1st, 2010 the system shall start sending monitoring reports to the administrator and write this event in the system log. This operation shall be repeated every 24 h.
(85, START, 1272661200000, system, send-monitoring-report-to-admin, monitoring-report, system-log, 86400000)

4. Detecting OAM&P requirement conflicts

4.1. Solution process

As we mentioned previously, the proposed method is based on the observed fact that requirement analysts still use natural language to write requirements. Our approach consists of providing OAM&P requirement analysts with a controlled natural language (CNL)-based framework to help them specify requirements in the most unambiguous and complete way. The framework also contains a program capable of detecting requirement inconsistencies. For this aim, the OAM&P requirement language defined in Section 3 is supposed to be

automated into a user interface where the requirement tuple structure is presented to the analyst in terms of the following EBNF:

< requirement > - >

ID: 0–9 {0–9}
S: START | STOP | FORBID
EVENT: < event > | VOID | 0–9 {0–9}
AGENT: < agent > | ALL
OPERATION: < operation >
INPUT: < object > {, < object > }
OUTPUT: < object >
HOLD: VOID | 0–9 {0–9}

The framework automatically builds the sets Obj, Agt, Opr, and Evt while the analyst is entering the requirement tuples. At the end of this step, the analyst is required to provide the relations between objects. The relations are the common object-oriented ones: inheritance (is-a relation), aggregation (has-a relation), and composition (contains-a relation). This is the first step in the OAM&P requirement conflict detection process. The whole process is depicted in Fig. 2.

4.2. Step 2: building object relation trees

Once requirements are written and object relations are provided, the framework interprets relations in terms of the following predicates:

- $h(\text{object}_1, \text{object}_2)$: object_2 is-a object_1
- $g(\text{object}_1, \text{object}_2)$: object_1 has-a object_2
- $c(\text{object}_1, \text{object}_2)$: object_1 contains-a object_2

The relations provided by the OAM&P analyst constitute a directed graph $G = (N, A)$, where N is the set of nodes (or vertices) and A is the set of arcs (or directed edges). The framework builds G by creating two nodes, object_i and object_j, and an arc from object_i to object_j for every $R(\text{object}_i, \text{object}_j)$, $R \epsilon \{h, g, c\}$. The next step for the framework is to derive all the predicates *reachable*$(\text{object}_i, \text{object}_j)$ that are true. Recall that the latter holds if there is a path in G that contains m nodes $\text{object}_1, \ldots, \text{object}_i, \ldots, \text{object}_j, \ldots, \text{object}_m$, $m \geqslant 2$. This problem is one of Depth First Search (DFS). In the general case, the specification provided by the analyst may result in multiple directed graphs G. The framework then has to execute DFS with every graph. For example, analyzing the graph depicted in Fig. 3 results in the following paths: FDA, FDB, GEB, HC, and KC.

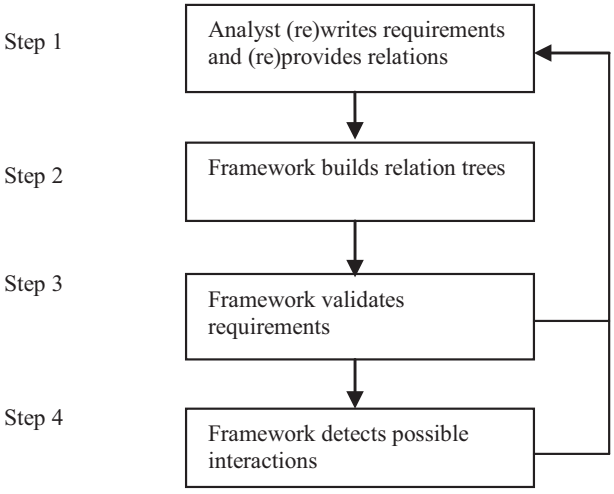

Figure 2 The proposed solution process.

4.3. Step 3: requirement statement validation

The framework then checks the validity of every requirement statement based on the following validation rules:

1. Every operation shall have an agent:(id, s, event, VOID, operation, object-set, object, hold) - > error(id)
2. Every operation shall have an input:(id, s, event, agent, operation, VOID, object, hold) - > error(id)
3. Every operation shall have an output:(id, s, event, agent, operation, object-set, VOID, hold) - > *error*(id)
4. The output of an operation shall be a single object:(id, s, event, agent, operation, object-set_1, object-set_2, hold) - > *error*(id)
5. If s equals FORBID or STOP, then hold must be equal to VOID(id, s, event, agent, operation, object-set, object, hold),

(s = FORBID OR s = STOP) AND hold ≠ VOID- > *error*(id)

1. If s equals STOP, then event must not be equal to VOID(id, s, event, agent, operation, object-set, object, hold),

s = STOP AND event = VOID- > *error*(id)

The first two validation rules are the same as in KAOS (Objectiver, 2007). The third one has been reported in Calisaya et al., 2008; Haibo et al., 2010; Van Lamsweerde et al., 1998a. After the framework checks the validity of the requirements in regard to the rules, displays the analysis results on the analyst's interface. The analyst repeats the process of rewriting erroneous requirement statements and re-launching the validity checking until there is no mistake (no inferred error(id)). Rules 1–3 and 5–6 relate to requirement specification mistakes. The analyst has to fix them. Solving the issue of rule 4 implies splitting the requirement into more than one so that each one has a single object as output.

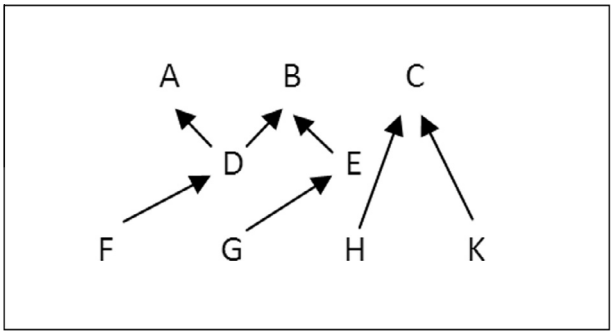

Figure 3 Path graph example.

4.4. Step 4: requirement interaction detection

In this step, the framework compares every pair of requirement tuples to detect conflicts. Conflicting tuples are reported to the analyst so that he can solve the conflicts by rewriting the requirements. He then has to re-launch the conflict detection. This process has to be executed as many times as needed until there is no requirement conflict (see Fig. 2). The conflict detection procedure uses the following conflict inference rules. Each rule corresponds to a specific type of conflict. For every type of conflict, we explain how the analyst should solve it.

4.4.1. Duplicated requirement (1)

The interaction is due to the two requirements being exactly the same or one being included in the other:

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $hold_2$),$id_1 \neq id_2$, $s_1 = s_2$, $event_1 = event_2$, $agent_1 = agent_2$ OR $agent_1$ = ALL, $operation_1 = operation_2$, $object\text{-}set_1 = object\text{-}set_2$, $object_1 = object_2$, $repeat_1 = repeat_2$, $hold_1 = hold_2$

-> *interaction*(id_1, id_2)

Resolution: The analyst should keep only one of the two requirements based on the customer's story.

4.4.2. Incompatible requirements (2)

In this type of conflict, the two requirements are ambiguous, incompatible, or contradictory.

- *Two operation frequencies* (2.1)

In this interaction, the same agent is required to perform the same operation on the same object but at two different frequencies. This interaction type has been called time discrepancies in Moser et al., 2011:

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $hold_2$),$id_1 \neq id_2$, $s_1 = s_2$ = START, $agent_1 = agent_2$ OR $agent_1$ = ALL, $operation_1 = operation_2$, $object_1 = object_2$ OR *reachable*($object_1$,$object_2$), $hold_1 \neq hold_2$

-> *interaction*(id_1, id_2)

Resolution: The analyst should check whether to keep the two requirements or only one based on the customer's story.

- *Start-forbid* (2.2)

In this conflict, the same event causes the same operation to be performed and to be forbidden, which is contradictory. This interaction type is also known as obstruction (Van Lamsweerde and Willemet, 1998b) and mutual exclusion (Calisaya et al., 2008):

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $hold_2$),$id_1 \neq id_2$, s_1 = START, s_2 = FORBID, $event_1 = event_2$, $agent_1 = agent_2$ OR $agent_1$ = ALL OR $agent_2$ = ALL, $operation_1 = operation_2$, $object\text{-}set_1 = object\text{-}set_2$, $object_1 = object_2$ OR *reachable*($object_1$, $object_2$) OR *reachable*($object_2$, $object_1$)

-> *interaction*(id_1, id_2)

Resolution: The analyst may have to correct or omit one or both of the requirements based on the customer's story.

- *Forbid-stop* (2.3)

This conflict is due to the same operation being stopped under a certain condition event and, at the same time, being unconditionally forbidden in another requirement:

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $hold_2$),$id_1 \neq id_2$, s_1 = FORBID, s_2 = STOP, $event_1$ = VOID AND $event_2 \neq$ VOID, $agent_1 = agent_2$ OR $agent_1$ = ALL OR $agent_2$ = ALL, $operation_1 = operation_2$, $object\text{-}set_1 = object\text{-}set_2$, $object_1 = object_2$

-> *interaction*(id_1, id_2)

Resolution: The analyst may have to correct or omit one or both of the requirements based on the customer's story.

- *Two condition events* (2.4)

This conflict is due to the same operation being executed, stopped, or forbidden on two different events:

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $repeat_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $repeat_2$, $hold_2$),$id_1 \neq id_2$, $s_1 = s_2$, $event_1 \neq event_2$ AND $event_1 \neq$ VOID AND $event_2 \neq$ VOID, $agent_1 = agent_2$ OR $agent_1$ = ALL, $operation_1 = operation_2$, $object\text{-}set_1 = object\text{-}set_2$, $object_1 = object_2$

-> *interaction*(id_1, id_2)

Resolution: Such a situation is not always problematic because there may be operations meant to be held under different conditions. The analyst should check if the conflict is really a specification mistake. To solve the interaction, the analyst may have to correct or omit one or both of the requirements based on the customer's story.

4.4.3. Assumption alteration (3)

This type of interaction holds when the output of one requirement's operation is part of the inputs (assumptions) or outputs (results) of the other's operation. In such a case, the alteration operated by the first requirement's operation on the inputs or outputs of the second requirement may be undesirable.

- *Input–output* (3.1)

This conflict holds between two requirements if one of them performs its operation on an object (output) that is an input in the other requirement. In telecommunication feature interactions, this situation is called assumption violation (Griffeth and Velthuijsen, 1994) because one feature could alter the assumption (input) of the other. In terms of our language, there is a conflict because the first requirement's output ($object_1$) is part of or contains, has, or specializes either directly or indirectly an object that is part of the second requirement's input set ($object\text{-}set_2$):

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $hold_2$),$id_1 \neq id_2$, $s_1 = s_2 = START$, $object_1 \in object\text{-}set_2$ OR $\exists$ o $\in$ $object\text{-}set_2$ and *reachable*($object_1$, o)

-> *interaction*(id_1, id_2)

Resolution: If the interaction is undesirable, the analyst should make sure that there is a requirement that protects the assumptions of one requirement from being altered by the other requirement.

- *Output–output* (3.2)

This conflict consists of the fact that one requirement alters the (or part of the) result (output) of the other:

(id_1, s_1, $event_1$, $agent_1$, $operation_1$, $object\text{-}set_1$, $object_1$, $hold_1$),

(id_2, s_2, $event_2$, $agent_2$, $operation_2$, $object\text{-}set_2$, $object_2$, $hold_2$),$id_1 \neq id_2$, $s_1 = s_2 = START$, $operation_1 \neq operation_2$, $object_1 = object_2$ OR *reachable*($object_1$, $object_2$)

-> *interaction*(id_1, id_2)

Resolution: This situation is not always undesirable because the result of one operation may be meant to be updated by another operation. The analyst should check this case. He may then have to make sure that there is a requirement that protects the results of one requirement from being altered by the other requirement.

4.4.4. Non-conflicting requirements (4)

The above detection inference rules are expressed in Table 1. This table is meant to be implemented in the framework and to be the basis of the detection procedure's operation. The latter looks in the table to trigger conflicting rules. For any pair of requirement tuples that cannot match any row in the table, the detection procedure will derive NOT *interaction*(). This is a conflict inference rule as well:$id_1 \neq id_2$, the two requirements (id_1, id_2) do not match with any row in Table 1.

-> NOT *interaction*(id_1, id_2)

The lines of Table 1 (conflict rules) are not mutually exclusive. Table 2 shows the sets of rules that can be triggered by the same couple of requirement tuples. This implies that a couple of requirement tuples might infer *interaction*() more than once. We suppose that the implementation of the requirement detection procedure displays the corresponding number of the interaction on the analyst's interface. The analyst then has to analyze the interactions and solve them.

4.5. OAM&P requirement specifications

The OAM&P requirement modeling language we presented in this article and the associated requirement validation rules (Section 4.3) and conflict inference rules (Section 4.4) allow to state OAM&P requirement specifications (RS). In the latter, the sets Obj, Agt, Opr, and Evt are specified by the analyst

Table 2 Jointly triggerable conflict rules.

R2.1, R2.4
R2.4, R3.2
R3.1, R3.2

Table 1 Conflict Inference Rules. Columns: 1: $id_1 = id_2$, 2: $s_1 = s_2$, 3: $s_1 = START$, 4: $s_1 = FORBID$, 5: $s_2 = START$, 6: $s_2 = STOP$, 7: $s_2 = FORBID$, 8: $event_1 = event_2$, 9: $event_1 = VOID$, 10:$event_2 = VOID$, 11:$agent_1 = agent_2$, 12: $agent_1 = ALL$, 13: $agent_2 = ALL$, 14:$operation_1 = operation_2$, 15: $obj\text{-}set_1 = obj\text{-}set_2$, 16:$object_1 = object_2$, 17:reachable($object_1$, $object_2$), 18: reachable($object_2$, $object_1$), 19: $object_1 \in object\text{-}set_2$, 20: $\exists$ o $\in$ $object\text{-}set_2$ AND reachable($object_1$, o), 21: $hold_1 = hold_2$. Cells: 1: true, 0: false, empty: true or false.

R	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	1						1			1			1	1	1					1
	0	1						1				1		1	1	1					1
2.1	0	1	1	0	1						1			1		1	0				0
	0	1	1	0	1						1			1		0	1				0
	0	1	1	0	1							1		1		1	0				0
	0	1	1	0	1							1		1		0	1				0
2.2	0	0	1	0	0	0	1	1			1			1	1	1					
	0	0	1	0	0	0	1	1			1			1	1	0	1				
	0	0	1	0	0	0	1	1			1			1	1	0		1			
	0	0	1	0	0	0	1	1				1		1	1	1					
	0	0	1	0	0	0	1	1				1		1	1	0	1				
	0	0	1	0	0	0	1	1				1		1	1	0		1			
	0	0	1	0	0	0	1	1					1	1	1	1					
	0	0	1	0	0	0	1	1					1	1	1	0	1				
	0	0	1	0	0	0	1	1					1	1	1	0		1			
2.3	0	0	0	1	0	1	0	0	1	0	1			1	1	1	0	0			
	0	0	0	1	0	1	0	0	1	0		1		1	1	1	0	0			
	0	0	0	1	0	1	0	0	1	0			1	1	1	1	0	0			
2.4	0	1						0	0	0	1			1	1	1					
	0	1						0	0	0		1		1	1	1					
3.1	0	1	1	0	1	0	0									0			1	0	
	0	1	1	0	1	0	0									0			0	1	
3.2	0	1	1	0	1	0	0							0		1	0				
	0	1	1	0	1	0	0							0		0	1				

through the Step 1–Step 4 process (Sections 4.1–4.4). The aim of an RS is to derive either *interaction*(id_1, id_2) or NOT *interaction*(id_1, id_2) for every pair of requirements (id_1, id_2). At the end of the process, no *error*() predicate is true in RS because there exist only well-formed requirement tuples. Additionally, RS contains *reachable*() and *interaction*() predicates.

Definition 4.1 (*Requirement specification*) A requirement specification (RS) consists of well-formed requirement tuples, *reachable*() predicates, and *interaction*() predicates.

An important aspect of RS is their correctness. An RS is correct if it is complete and consistent. Consistency means that for each couple of requirements, either *interaction*() or NOT *interaction*() is derived but not both. Completeness means that for every couple of requirements, a verdict, either *interaction*() or NOT *interaction*(), exists.

Definition 4.2 (*Completeness and consistency of RS*) An RS is complete if for each pair of requirement tuples, at least *interaction*() or NOT *interaction*() is true. An RS is consistent if for each couple of requirement tuples, *interaction*() and NOT *interaction*() cannot both be true.

Proposition 4.1. An RS specified using the here defined language and the Steps 1–4 process is complete.

Proof. Table 1 contains all the requirement conflict inference rules. In an RS, the numbers of requirements, requirement parameters (eight parameters: id, s-value, and event) and *reachable*() predicates are finite. Additionally, according to Table 1, triggering any row (conflict rule) infers *interaction*(). According to conflict inference rule 4, a pair of requirements that cannot match with any row of Table 1 results in inferring NOT *interaction*(). Hence, any RS specified using the here defined language and process is complete. □

Proposition 4.2. An RS specified using the here defined language and the Steps 1–4 process is consistent.

Proof. As stated in the proof of Proposition 1, for every pair of requirements, either *interaction*() or NOT *interaction*() is derived. According to Table 1, Table 2, and the definition of conflict inference rule 4, a couple of requirements triggers either rule 4, which sets *interaction*() to false, or triggers one or more of the rows of Table 1, which set *interaction*() to true once or many times. Hence, any RS specified using the here defined language and process is consistent. □

4.5.1. Computational complexity

Let Obj and Req be the sets of objects and requirements, respectively. Step 1 of Fig. 2 is a manual one. Steps 2–4 are fully automatable. Step 2 can be divided into two. First, the framework builds the relation trees. The complexity of this operation is $O(|Obj|^2)$. Then, the framework writes a list of all the reachable() predicates that are true, which corresponds to a DFC problem. The latter's complexity equals O(|Obj|) (Heineman et al., 2006). The complexities of step 3 and step 4 are O(|Req|) and $O(|Obj|^2.|Req|^2)$, respectively. Thus, the computational complexity of the whole process is $O(|Obj|^2.|Req|^2)$. If we assume that |Req| > |Obj|, the complexity approaches $O(|Req|^4)$. However, the conflict detection procedure should be implemented so that it checks only the rows of Table 1 then immediately concludes NOT interaction() if none of those rows apply. This significantly reduces the computation complexity. This has been proven through simulations conducted to study the scalability of the proposed solution. Further details can be found in Section 6.

5. Related work

Aspects of the here studied problem have been addressed by Chentouf (2012) and KAOS (Objectiver, 2007).

5.1. The work of Chentouf

(Chentouf, 2012) presents research on handling OAM&P service interactions. The approach consists of modeling each OAM&P feature by abstracting its action as either *use* or *modify*. In terms of the here presented language, Chentouf, 2012 models each OAM&P feature as one of the followings:

- Agent *modify* object
- Agent *use* object

The operation *modify* abstracts any operation that actually modifies data or affects processing. For example, write, delete, and execute. The operation *use* abstracts operations that do not change data or affect processing, such as read, has, and apply on. The concern of this simple language goes to modeling service logic rather than requirements. The language defined in the current paper aims to model requirements.

Another relevant aspect is the interaction detection procedure. The emphasis in Chentouf, 2012 is placed on detecting OAM&P feature interactions. The present work addresses OAM&P requirement conflicts. In the former work, OAM&P feature interactions are defined to hold between two features if one of the following situations is encountered:

- $agent_1$ *use* object; $agent_2$ *modify* object, $agent_1 \neq agent_2$.
- $agent_1$ *modify* object; $agent_2$ *modify* object, $agent_1 \neq agent_2$.

The first FI pattern is equivalent to our requirement conflict rule input–output (3.1). The second pattern is equivalent to the rule output–output (3.2).

Let us compare the reliability of the two approaches. The fact that the language of Chentouf, 2012 does not contain concepts such as forbid, stop, and event makes it less expressive. More precisely:

- In Chentouf, 2012, $agent_1$ must be different from $agent_2$, and the agent ALL is not defined. This makes all cases of interactions 1 and 2.1–2.4 and some cases of 3.1–3.2 undetectable, according to Table 1, columns 11–13.
- The predicate *reachable* is not expressed, which prevents some conflicts of types 2.1–2.3 and all conflicts of 3.1–3.2 from being detected, based on Table 1, columns 17, 18, and 20.
- The *hold* parameter is not expressed, which means that conflict 2.1 is not detectable, based on Table 1, column 21.

The approach of Chentouf, 2012 does not detect any of the conflict examples presented in Section 6 except the example of

conflict 3.1. Using the language of Chentouf, 2012, the latter example can be written:

- Broker *modify* rate-table
- Administrator *use* rate-table

These two requirements correspond to the pattern:$agent_1$ *use* object; $agent_2$ *modify* object, $agent_1 \neq agent_2$.

Another advantage of the here proposed work is the fact that it defines a complete framework that guides analysts to cope with OAM&P requirement conflicts.

5.2. The requirement modeling language KAOS

KAOS defines a requirement conflict in the situation where the satisfaction of one goal prevents the satisfaction of another. The relation between the two requirements is called an obstacle (Objectiver, 2007). However, KAOS does not contain any defined conflict detection rule or detection procedure. The main focus of KAOS is placed on modeling requirements. To detect requirement conflicts, we had to modify KAOS. We ended up with a new language.

Table 4 summarizes the differences between the here defined language and KAOS. As can be seen in this table, KAOS offers a graphical notation, which is not defined in the present work. Based on this notation, KAOS provides different abstraction views. For example, a responsibility model is part of the requirement graphical model that focuses on a given agent. Similarly, an operation model isolates a given operation and its related inputs and outputs. However, these two features that are not defined in the present work can be seen as relevant to the implementation of the language and framework more than to the design of the underlying modeling approach and conflict detection solution.

6. Case study

6.1. Selected OAM&P requirement conflict examples

The following are some selected OAM&P conflict examples. We suppose that the identifiers of the two requirements are not equal, and we do not write them in the examples.

6.1.1. Duplicated requirement (1)

Requirement:

- The system shall authenticate incoming ITSP calls using the user-ID.

Requirement tuples:
(START, ITSP-call-received, system, authenticate, ITSP-user-ID, received-call, VOID)
(START, ITSP-call-received, system, authenticate, ITSP-user-ID, received-call, VOID)

This couple of tuples triggers conflict rule 1. The conflict consists of the fact that the same requirement is written twice with different ID numbers.

6.1.2. Two operation frequencies (2.1)

Requirements:

- The system shall invoice broker accounts once per semester using CDR.
- The system shall invoice subscriber accounts once per month using CDR.

Relations:

- Broker accounts contain subscriber accounts.

Requirement tuples:
(START, VOID, system, invoice, {broker-CDR, broker-account}, broker-account, 15811200000)
(START, VOID, system, invoice, {subscriber-CDR, subscriber-account}, subscriber-account, 2592000000)*reachable*(broker-account, subscriber-account)

This couple of tuples triggers conflict rule 2.1. The interaction here means that the second requirement is included in the first one (a broker account contains subscriber accounts), and the two requirements have different frequencies. According to Table 2, couples of requirements that trigger rule 2.1 might trigger rule 2.4. This example does not because both of the events are equal to VOID (see Table 1).

6.1.3. Start-forbid (2.2)

Requirements:

- It shall be forbidden to delete invoices.
- A broker should be able to delete his own subscribers' accounts.

Relations:

- A subscriber account contains invoices.

Requirement tuples:
(FORBID, VOID, ALL, delete, invoice, invoice, VOID)
(START, VOID, broker, delete, subscriber-account, subscriber-account, VOID)*reachable*(subscriber-account, invoice)

This couple of tuples triggers rule 2.2. The interaction holds because the agent broker is included in ALL, a subscriber account contains invoices, the two operations are the same, but one of them is forbidden and the other is commanded.

6.1.4. Forbid-stop (2.3)

Requirements:

- It shall be forbidden for PBX accounts to have virtual circuits.
- The system shall deactivate virtual circuits for PBX accounts that do not pay for the service.

Requirement tuples:
(FORBID, VOID, system, activate-virtual-circuits, PBX, PBX, VOID)

(STOP, unpaid-virtual-circuit-service, system, activate-virtual-circuits, PBX, PBX, VOID)

These tuples trigger rule 2.3. There is a conflict because the same operation is unconditionally forbidden by one requirement, whereas the other requirement is meant for stopping it. These two requirements are incompatible.

6.1.5. Two condition events (2.4)

Requirements:

- If the service plan is elapsed, the system should block no more authorized calls.
- If the credit limit is reached, the system shall block no more authorized calls.

Requirement tuples:

(STOP, service-plan-elapsed, system, block, unauthorized-calls, unauthorized-calls, VOID)

(STOP, credit-limit-reached, system, block, unauthorized-calls, unauthorized-calls, VOID)

Rule 2.4 is triggered. There is an interaction because the same agent is required to perform the same operation but on two different events. According to Table 2, couples of requirements that trigger rule 2.4 might trigger rule 3.2 as well. This example does not because both s_1 and s_2 are not equal to START (see Table 1).

6.1.6. Input–output (3.1)

Requirements:

- The broker shall be able to access the rate table.
- The administrator shall create an access list based on the rate table.

Requirement tuples:

(START, VOID, broker, access, rate-table, rate-table, VOID)

(START, VOID, administrator, create, rate-table, access-list, VOID)

Rule 3.1 is triggered by these two requirements. The conflict is due to the fact that the assumption of the second requirement (input: rate-table) is equal to the output of the first requirement. According to Table 2, requirements that trigger rule 3.1 might trigger rule 3.2 too. This example does not because the outputs of the two requirements are not equal (see Table 1).

Table 3 Simulation results.

Requirements	Time (s)	Time (min)
100	5	0.1
300	5	0.1
1000	5	0.1
3000	7	0.1
5000	10	0.2
10,000	24	0.4
30,000	148	2.5
60,000	816	13.6
200,000	3680	61.3

6.1.7. Output–output (3.2)

Requirements:

- The system shall use number filters to modify number prefixes
- Subscribers shall be able to create virtual numbers

Relations:

- Virtual numbers contain number prefixes

Requirement tuples:

(START, VOID, system, modify, {number-prefix, number-filter}, number-prefix, VOID)

(START, VOID, subscriber, create, virtual-number-list, virtual-number, VOID)*reachable*(virtual-number, number-prefix)

These requirements trigger rule 3.2. There is a conflict because the output of the second requirement contains the output of the first one.

6.2. Scalability tests

To study the scalability of the proposed solution, a requirement conflict detection program based on Table 1 has been implemented in Java. A set of 50 requirements have been written, and the 25 object couples have c, g, or h relations. To feed the test with more requirements, a set of 50 requirements has been duplicated as needed. The program has been written so

Table 4 Differences between KAOS and the proposed language.

Feature or concept	Present work	KAOS
Goal	No	Yes
Operation	Yes	Yes
Agent	Yes	Yes
Input	Yes	Yes
Output	Yes; one object only	Yes; there may be more than one object
Start, stop, forbid	Yes	Start and stop only
c(),*g*(),*h*()	Yes	Yes; not considered for conflict detection
Hold	Yes	No
Defined process through a framework	Yes	Yes
Conflict detection	Yes	No
Multiple abstraction views	No	Yes
Graphical notation	No	Yes

that every conflict detection iteration goes through all of the rows of Table 1, even if a row has been matched before reaching the end of the table. This means that the results of a real exploitation of the solution will certainly be better than the ones reported in this paper. Table 3 contains the simulation results.

The results show that the proposed solution requires an acceptable computation time, which remains less than a minute for more than 10,000 requirements. The results also show that the solution scales very well as the number of requirements increases. We do not know how many requirements could be written for an OAM&P implementation project. However, based on our previous industrial experience, we can say that an OAM&P implementation can contain approximately 200000 LOC. We conjecture that the number of requirements is always less than the number of LOC, which is why we tested the framework implementation's scalability with this number of requirements despite it seeming exaggerated.

7. Conclusion

This article has addressed the problem of detecting OAM&P requirement conflicts. The aim is to solve them before they translate into software defects. We have proposed a solution composed of an OAM&P feature modeling language, a set of conflict detection inference rules, and a complete work process meant to be implemented by OAM&P requirement analysts and service engineers. The advantage of the proposed language is that it is presented in a controlled natural language for easy by analysts and engineers. Completeness, consistency, and scalability of the proposed solution are proven. We demonstrate these contributions using proof-of-concept examples and scalability simulation tests. Other contributions of the present work consist of determining the OAM&P requirement conflict causes and identifying and classifying their types.

Improvements on this work are still possible. Our conflict detection rules only handle pair-wise conflicts. We will try to generalize the detection procedure so it can detect n-way interactions that may involve more than two requirement tuples. However, more complex rules would be required. In addition, our solution is based on a syntactical comparison between requirement tuples. Therefore, it cannot detect semantic conflicts, for example, when the same object or the same operation is expressed by the analyst by means of two different words. Addressing this point should open the way for the proposed solution to detect more subtle conflicts.

References

Amer, A., Karmouch, A., Gray, T., Mankovskii, S., 2000. Feature interaction resolution using fuzzy policies. In: Calder, M., Magill, E. (Eds.), Feature Interactions in Telecommunications and Software Systems V. IOS Press, Amsterdam, pp. 94–112.

Amyot, D., Logrippo, L. (Eds.), 2003. Feature Interactions in Telecommunications and Software Systems VI. IOS Press, Amsterdam.

Amyot, D., Charfi, L., Corse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., Ware, T., 2000. Feature description and feature interaction analysis with use case maps and LOTOS. In: Calder, M., Magill, M. (Eds.), Feature Interactions in Telecommunications and Software Systems V. IOS Press, Amsterdam, pp. 274–289.

Blom, J., 1997. Formalisation of requirements with emphasis on feature interaction detection. In: Dini, P., Boutaba, R., Logrippo, L. (Eds.), Feature Interactions in Telecommunications and Software Systems III. IOS Press, Amsterdam, pp. 61–77.

Brachman, R.J., Levesque, H.J. (Eds.), 1985. Readings in Knowledge Representation. Morgan Kaufmann, Boston.

Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S., 2003. Feature interaction: a critical review and considered forecast. Comput. Networks 41 (1), 115–141.

Calder, M., Magill, E. (Eds.), 2000. Feature Interactions in Telecommunications and Software Systems V. IOS Press, Amsterdam.

Calisaya, E.S., Borges, M.R.S., Campos, M.L.M., 2008. Automatic discovery of interactions between software requirements. In: The 20th International Conference on Software Engineering and Knowledge Engineering, San-Francisco.

Charcranoon, S., El-Bawab, T.S., Shin, J.D., Cankaya, H.C., 2005. Framework for Operation and Maintenance (OAM) in Optical Burst Switched Networks. J. Network Syst. Manage. 13 (4), 387–408.

Charnois, T., 1997. A natural language processing approach for avoidance of feature interactions. In: Dini, P., Boutaba, R., Logrippo, L. (Eds.), Feature Interactions in Telecommunications and Software Systems III. IOS Press, Amsterdam, pp. 347–363.

Chentouf, Z., Khoumsi, A., 2013. A high abstraction level approach for detecting feature interactions between telecommunication services. J. King Saud Univ. Comput. Inf. Sci. 25 (1), 99–115.

Chentouf, Z., 2012. Detecting OAM&P design defects using a feature interaction approach. Int. J. Network Manage. 22 (2), 95–103.

Chentouf, Z., Cherkaoui, S., Khoumsi, A., 2003a. Implementing online feature interaction detection in SIP environment. In: 10th International Conference on Telecommunications, Tahiti.

Chentouf, Z., Cherkaoui, S., Khoumsi, A., 2003b. Feature Interaction detection in SIP environment. Telecommunication Syst. 24 (2), 251–274.

Chentouf, Z., Cherkaoui, S., Khoumsi, A., 2003c. New management methods for feature and preference interactions. In: IFIP/IEEE International Conference on Management of Multimedia Networks and Services, Belfast.

Cherkaoui, S., Khoumsi, A., 2002. Mobile and static agents for service interactions resolution in telecommunication environments. In: 9th IEEE International Conference on Telecommunications (ICT'2002), Beijing.

Chi, C., Wang, D., Hao, R., 2003. A framework on feature interactions in optical network protocols. In: Amyot, D., Logrippo, L. (Eds.), Feature Interactions in Telecommunications and Software Systems VI. IOS Press, Amsterdam, pp. 55–69.

Dardenne, A., Van Lamsweerde, A., Fickas, S., 1993. Goal directed requirements acquisition. Sci. Comput. Program 20 (1–2), 3–50.

Darimont, R., Delor, E., Massonet, P., Van Lamsweerde, A., 1998. GRAIL/KAOS: An environment for goal-driven requirements engineering. In: Proceedings of the 20th Int. Conference on Software Engineering, Kyoto.

Darimont, R., Van Lamsweerde, A., 1996. Formal refinement patterns for goal-driven requirements elaboration. In: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco.

Dini, P., Boutaba, R., Logrippo, L. (Eds.), 1997. Feature Interactions in Telecommunications and Software Systems III. IOS Press, Amsterdam.

Donadio, P., Paparella, A., Berde, B., 2009. Service-oriented technology for TMN-based network management services. Bell Labs Tech. J. 14 (1), 161–172.

Dubois, E., Hagelstein, J., Rifaut, A., 1988. Formal Requirements Engineering with ERAE. Philips J. Res. 43 (3), 393–414.

Du Bousquet, L., Richier, J.L. (Eds.), 2007. Feature Interactions in Telecommunications and Software Systems VIII. IOS Press, Amsterdam.

Gammelgaard, A., Kristensen, J.E., 1994. Interaction detection, a logical approach. In: Bouma, L.G., Velthuijsen, H. (Eds.), Feature Interactions in Telecommunications and Software Systems I. IOS Press, Amsterdam, pp. 178–196.

Georgatsos, P., Nauta, T., Velthuijsen, H., 1997. Role of service management in service interaction handling in an IN environment. In: Dini, P., Boutaba, R., Logrippo, L. (Eds.), Feature Interactions in Telecommunications and Software Systems III. IOS Press, Amsterdam, pp. 213–225.

Gervasi, V., Zowghi, D., 2005. Reasoning about Inconsistencies in Natural Language Requirements. ACM Trans. Software Eng. Methodol. 14 (3), 277–330.

Gibson, P., Mery, D., 1997. Telephone feature verification: translating SDL to TLA+. In: Eighth SDL Forum (SDL'1997), Evry, France.

Greenspan, S., Mylopoulos, J., Borgida, A., 1994. On formal requirements modeling languages. In: Proceedings of the 16th Int. Conf. Software Eng, Los Alamitos.

Greenspan, S.J., Borgida, A., Mylopoulos, J., 1986. A requirements modeling language and its logic. Inf. Syst. 11 (1), 9–23.

Griffeth, N.D., Velthuijsen, H., 1994. The negotiating agents approach to runtime interaction resolution. Feature interactions in telecommunications systems. In: Bouma, L.G., Velthuijsen, H. (Eds.), Feature Interactions in Telecommunications and Software Systems I. IOS Press, Amsterdam, pp. 217–235.

Haibo, H., Yang, D., Ye, C., Fu, C., Li, R., 2010. Detecting Interactions between Behavioral Requirements with OWL and SWRL. World Acad. Sci. Eng. Technol. 48 (1), 330–336.

Heineman, G.T., Pollice, G., Selkow, S., 2006. Algorithms in a Nutshell, O'Reilly, 2009.

Ilić, D., Troubitsyna, E., Laibinis, L., Leppänen, S., 2006. Formal verification of consistency in model-driven development of distributed communicating systems and communication protocols. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Cyprus.

Jureta, I.J., Borgida, A., Ernst, N.A., Mylopoulos, J., 2010. Techne: towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In: The 18th IEEE Requirement Eng. Conf, Australia.

Kimbler, K., Bouma, L.J. (Eds.), 1998. Feature Interactions in Telecommunications and Software Systems IV. IOS Press, Amsterdam.

Kolberg, M., Magill, E., 2001. Handling incompatibilities between services deployed on IP-based networks. In: IEEE Intelligent Networks, Boston.

Koymans, R., 1992. Specifying Message Passing and Time-Critical Systems with Temporal Logic. Springer-Verlag, Berlin.

Lavinal, E., Desprats, T., Raynaud, Y., 2009. A multi-agent self-adaptive management framework. J. Network Manag. 19 (3), 217–235.

Lennox, J., Schulzrinne, H., 2000. Feature interaction in Internet telephony. In: Calder, M., Magill, E. (Eds.), Feature Interactions in Telecommunications and Software Systems V. IOS Press, Amsterdam, pp. 38–50.

Mariangela, L., Pierluigi, I., 2004. Market research for requirements analysis using linguistic tools. Requirement Eng. 9 (1), 40–56.

McKiou, M., Buckley, F., 2003. Data-driven fault management within a distributed object-oriented OAM&P framework. Bell Labs Tech. J. 8 (1), 157–179.

Modarresi, A., Mohan, S., 2000. Control and management in next generation networks: challenges and opportunities. IEEE Commun. Mag. 38 (10), 94–102.

Moser, T., Winkler, D., Heindl, M., Biffl, S., 2011. Automating the detection of complex semantic conflicts between software requirements. In: The 23rd International Conference on Software Engineering and Knowledge Engineering, Miami.

Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M., 1990. Telos: representing knowledge about information systems. ACM Trans. Inf. Syst. 8 (4), 325–362.

Nakamura, M., Reiff-Marganiec, S. (Eds.), 2009. Feature Interactions in Telecommunications and Software Systems X. IOS Press, Amsterdam.

Nakamura, M., Kakuda, Y., Kikuno, T., 1997. Petri-net based detection method for non-deterministic feature interactions and its experimental evaluation. In: Dini, P., Boutaba, R., Logrippo, L. (Eds.), Feature Interactions in Telecommunications and Software Systems III. IOS Press, Amsterdam, pp. 138–152.

Objectiver, 2007. A KAOS Tutorial. [pdf] Paris. Available at <http://www.objectiver.com> [Accessed 3 September 2013].

Reiff-Marganiec, S., Ryan, M. (Eds.), 2005. Feature Interactions in Telecommunications and Software Systems VIII. IOS Press, Amsterdam.

Rizzo, M., Garyfalos, A., 2000. Using SIP to negotiate over user requirements in personalized internet telephony services. In: Proceedings of the SIP 2000, Paris.

Silverman, K., Brenner, M., Shannon, G., 2000. Toward a vision for network and service management. Bell Labs Tech. J. 5 (4), 21–30.

Van Lamsweerde, A., 2001. Goal-oriented requirements engineering: a guided tour. In: Proc. 5th IEEE Int. Symposium on Requirements Eng, Toronto.

Van Lamsweerde, A., 2000a. Requirements engineering in the year 00: a research perspective. In: Proceedings of the 22nd International Conference on Software Engineering, New York.

Van Lamsweerde, A., Letier, E., 2000b. Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Software Eng. 26 (10), 978–1005.

Van Lamsweerde, A., Darimont, R., Letier, E., 1998a. Managing conflicts in goal-driven requirements engineering. IEEE Trans. Software Eng. 24 (11), 908–926.

Van Lamsweerde, A., Willemet, L., 1998b. Inferring Declarative Requirements Specifications from Operational Scenarios. IEEE Trans. Software Eng. (Special issue on scenario management).

Van Lamsweerde, A., Darimont, R., Massonet, P., 1995. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learned. In: Proceedings of the 2nd Int. Symp. On Requirements Engineering, New York.

Yu, E., 1997. Towards modeling and reasoning support for early requirements engineering. In: Proc. 3rd IEEE Int. Symposium on Requirements Eng, Annapolis.