SOCA
DOI 10.1007/s11761-010-0062-7

SPECIAL ISSUE PAPER

MaDe4IC: an abstract method for managing model dependencies in inter-organizational cooperations

Lianne Bodenstaff · Andreas Wombacher · Manfred Reichert · Roel Wieringa

Received: 4 March 2010 / Revised: 17 May 2010 / Accepted: 31 May 2010

Abstract Inter-organizational cooperations are complex in terms of coordination, agreements, and value creation for involved partners. When managing complex cooperations, it is vital to maintain models describing them. Changing one model to regain consistency with the running system might result in new inconsistencies. As a consequence, this maintenance phase grows in complexity with increasing number of models. In this context, challenges are to ensure consistency at design time and to monitor the system at runtime, i.e., at design time, consistency between different models describing the cooperation needs to be ensured. At runtime, behavior of the software system needs to be compared with its underlying models. In this paper, we propose a structured and model-independent method that supports ensuring and maintaining consistency between running system and underlying models for inter-organizational cooperations.

This research has been supported by the Dutch Organization for Scientific Research (NWO) under contract number 612.063.409.

L. Bodenstaff (✉) · A. Wombacher · R. Wieringa
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: l.bodenstaff@utwente.nl

A. Wombacher
e-mail: a.wombacher@utwente.nl

R. Wieringa
e-mail: r.j.wieringa@utwente.nl

M. Reichert
University of Ulm, James-Franck-Ring, 89069 Ulm, Germany
e-mail: manfred.reichert@uni-ulm.de

Keywords Consistency · Inter-organizational models

1 Introduction

Model-based approaches are used in various fields in computer science like software development, information systems engineering, and e-business development [23,31,55]. Many of these approaches support *several* models, each emphasizing one specific aspect or part of the described software system. In this paper, we consider such model-based approaches for realizing *inter-organizational cooperations*. For example, we consider Web service compositions and e-business cooperations. Both are often complex in terms of coordination, agreements and value creation for the involved partners.

Due to the complex nature of inter-organizational cooperations, usually, a variety of models is used to specify the information system to be developed. For example, financial benefits are captured in a business model [39], while coordination details are specified in a process model (see [11] or [13]). Using several models to represent one complex information system has many advantages. Especially, model understandability is enhanced since each model only represents some of the information about the system to be developed. As a result, complexity is reduced for decision makers interpreting the models as well as for engineers developing and maintaining them; especially for cooperations in which different partners with different business goals need to come to an agreement, such multi-model approach is beneficial. As typical example of inter-organizational cooperation consider product and change management where different partners need to agree on a particular product change. For example, an automotive vendor and its suppliers need to agree on changes in the design of a car [54].

Published online: 19 June 2010

1.1 Problem statement

Although using several models to represent one complex system enhances usability when developing a specific model, new challenges arise. Modeling complexity is reduced by developing several models at design time, but these models together form the basis for the running cooperation, and in the end the running information system; i.e., system implementation must represent the combination of these different models. Therefore, it is of fundamental importance that the different models describe the same system, i.e., they are *consistent* with each other—this constitutes our challenge. The challenge is to *ensure consistency* between different models describing one system before implementation. We refer to this as the problem of ensuring *design time consistency*.

If the different models describing a particular cooperation are consistent with each other, there is a proper basis for implementing the information system. However, at runtime the behavior of the system might be different than agreed upon. Such deviant behavior can be caused by implementation errors. Another major cause is partners in the cooperation that do not behave according to the agreement. For example, business partners might not pay in time, or agreed upon response times are violated. Furthermore, deviant behavior might be caused by events that cannot be controlled by the business partners. Usually, such behavior (e.g., customer behavior) can be merely estimated when developing the models. In all these cases the running system behaves differently from the agreed upon models, i.e., the running system and the models describing it are inconsistent. We refer to this as *runtime inconsistency*. The latter is not always problematic, but typically some action is taken when inconsistencies occur. The challenge is to *monitor* the system such that inconsistencies with models it relies on can be detected.

Furthermore, when managing complex cooperations, it is vital to maintain the models describing them to keep an overview on the behavior and successfulness of the cooperation. Especially this *maintenance phase* is challenging since the different models are tightly connected and describe different perspectives of the same system. Changing one model to regain consistency with the running system might result in new inconsistencies between the different models. As a consequence, this maintenance phase of the models is time consuming and grows in complexity with increasing number of models describing the system.

The problem of checking consistency between related models, of checking consistency between a running system and its underlying models, and of managing the running system by maintaining consistency between models and running system is not new. Concerning design time consistency, there exist multi-model approaches, for example, Unified Modeling Language (UML) [32] and Open Distributed Processing (ODP) [9]. Both aim at consistent model development. However, these approaches are *model-specific* and are not applicable to other modeling languages. Especially when using different modeling languages that are not directly related, developers do not have such support. Furthermore, there exist some approaches that support multi-model development, but they stay on a high level explaining *what* should be done rather than *how* this should be accomplished [37,48].

Concerning runtime consistency, there exist monitoring approaches that support consistency checking of the running system and the models describing it (e.g., [44,46]). However, these approaches mainly focus on monitoring the running system against *one* model, neglecting dependencies between this model and others. Furthermore, the most challenging and dynamic part of the problem, i.e., maintaining consistency between models and running system, is even less supported in such consistency approaches.

1.2 Contribution

The main problem in ensuring and maintaining consistency between a set of models is that these models are *interrelated* and, therefore, changing one model might affect several other models. Therefore, the main challenge is to first identify the exact nature of relations between the models, and second to identify effects changes in one model have on the others. In this context, this paper proposes a method that supports ensuring and maintaining consistency between models and running system for inter-organizational cooperations. Our goal is to provide a *structured* and *model-independent approach* to check and maintain consistency.

In Sect. 2 we position our research by discussing the conceptual frame and by providing a categorization of models and consistency. We continue in Sect. 3 with a thorough problem analysis for ensuring and maintaining consistency in inter-organizational models. The state of the art is reviewed in the light of this analysis in Sect. 4. Section 5 introduces our comprehensive MaDe4IC method for MAnaging DEpendencies in Inter-organizational Cooperations. We discuss in Sect. 6 how we evaluate our method. We conclude this paper with a summary in Sect. 7.

2 Terms used in our MaDe4IC method

The proposed approach is model independent. Since related work is usually based on a specific model providing references on used terms is misleading. Detailed information and references to the individual terms are available in Sect. 4.

2.1 Inter-organizational cooperation

In this paper, we refer to inter-organizational cooperations where a cooperation is some voluntary interaction between

two or more partners. Such cooperation can be short term and market based, but also long term and of collaborative nature. Models describing these cooperations are typically *conceptual models* but can be expressed in a variety of languages like UML Activity Diagrams [38], Petri Nets [27], and BPMN [56]. An inter-organizational model as addressed in our method is a conceptual model that focuses on *exchanges* (i.e., interactions) between different partners. These kinds of models omit representing *internal* behavior of the partners involved. Furthermore, it assumes that communication and exchange of information between partners is (partly) dependent on *information technology*. Typically, such information technology enabled business models are referred to as e-commerce business models (cf. [50]).

2.2 Consistency

Models of inter-organizational cooperations are used for consistency *checking* at design time as well as *maintaining* consistency during runtime of the system. Consistency is checked and maintained *across* different models [1] and *within* models [43]. Consistency can be defined in many ways. Classical Aristotelian logic provides us with a semantic notion of consistency [47]:

> "Two or more statements are considered to be *consistent* if they are simultaneously true under some interpretation."

In modern logic the syntactic notion for consistency is defined as follows [30]:

> "A set of statements is considered to be consistent to a certain logical calculus if no formula $P \wedge \neg P$ can be derived from those statements by the rules of the calculus, i.e., the statements are free from contradictions."

Therefore, we define the term 'consistency' as the *absence of contradictions*. At design time we distinguish between consistency within models (i.e., intra-model consistency) and consistency across models (i.e., inter-model consistency). Consistency is always determined under some interpretation. A schematic view on the different consistency checks considered in this paper is given in Fig. 1.

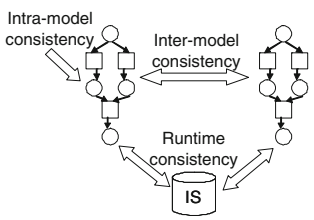

Fig. 1 Intra-model, inter-model, and runtime consistency relations

For *intra-model consistency*, we assume that the interpretation under which consistency is determined is given by the definition of the model-specification language. The produced model needs to be syntactically well formed and meaningful, i.e., consistent with the specification. Aside from the official specification, additional constraints on the models can be formulated in a specific context. For example, one might want to reduce expressiveness to avoid complex models. Here, we assume provided models are intra-model consistent (i.e., *well-formed*) with respect to their specification.

The challenge in checking *inter-model consistency* at *design time* is defining the proper interpretation under which these models are considered being consistent. Especially for models defined on different levels of abstraction, or defined in different modeling languages, this is not a straightforward exercise.

During *runtime*, first consistency is *checked* between the running system and an interpretation of each model. Again, defining this interpretation is a challenge. As a second step, consistency can be *maintained* by adapting models or implementations when contradictions are detected.

2.3 Categorization for models and consistency

Based on the above terms and the study of related work (Sect. 4), in the following a categorization of different approaches is given. We distinguish between the *type of models* which is considered, the *type of consistency* which is ensured, and the way consistency is *checked* through the different approaches.

Type of models. We distinguish approaches which handle consistency between different *viewpoints* on a system and approaches which handle consistency between different *partial models* of a system. A *viewpoint* on a system describes the entire system under development and focuses on a specific characteristic (e.g., the messages exchanged between partners). Reduction of complexity in modeling the system is accomplished by leaving out those aspects of the system that do not belong to the viewpoint characteristic. For example, one viewpoint might be the cost perspective of the cooperation, while another one describes the order in which messages are exchanged. As opposed to viewpoint models, *partial models* describe different parts of the system in separate models. To reduce complexity, the system under development is divided into parts. For example, a company develops separate models for each partner it interacts with.

The distinction between these two approaches is important since it influences the consistency relation between the models. Different viewpoints have a complete overlap in the modeling *domain* while their *focus* is disjoint. The challenge is to find the exact relation between the different foci. Partial models might have an overlap in the domain, but this is not a

complete overlap. The focus of the models, however, might be equal. For example, two Entity-Relationship diagrams of which each one describes a part of the system, have the same focus. In this case, the challenge is to find the relation between the partial models rather than to find the relation between the foci.

Since conceptual models that are used for modeling, inter-organizational cooperations can be both viewpoints and partial models, we look for an approach that checks consistency for both types.

Type of consistency. We distinguish different types of consistency. *Intra-model consistency* considers well-formedness of a model. The interpretation used for determining consistency is according to the requirements set for the specification language. *Inter-model consistency* checks consistency *between* two or more models. The interpretation used for determining consistency depends on the type of model used and on restrictions set by the engineer. However, in this paper, two models are considered as being consistent with each other if a specification can be found which represents both models. *Homogeneous* (i.e., *intra-language*) approaches consider models of the same type, while *heterogeneous* (i.e., *inter-language*) approaches enable consistency checks between models expressed in different languages.

For modeling inter-organizational cooperations, typical heterogeneous models are used. Therefore, we look for an approach that handles such heterogeneity.

Ensuring consistency. Further, we distinguish between different ways of *ensuring* consistency. Two main options are to *check* consistency after models are developed or to *ensure* consistency by construction during the development process. Checking consistency can be done by *testing* the models with some model checker, or by finding a *translation*. Usually models are translated into a semantically well-defined formalism which allows for formal consistency checking. When translating models, either they are completely translated or only the overlapping parts between them are translated. A complete translation is time consuming, while in a partial translation the overlap between models is first determined. Especially when dealing with heterogeneous models this is not straightforward. When consistency is ensured during model *construction*, either additional development requirements for the models are set or consistency is defined by relating their meta-models.

Aside from consistency checking, we aim at maintaining consistency during runtime of the system through some adaptations. Maintaining consistency cannot be done through construction since this is accomplished at design time. Maintaining consistency is done at runtime. Therefore, our approach should allow consistency ensuring through checking rather than through construction.

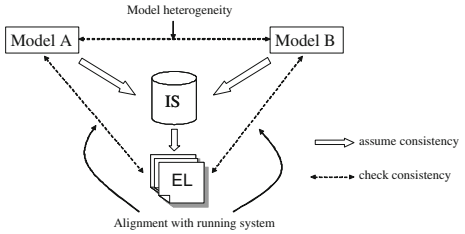

Fig. 2 Consistency relations between models, event logs, and information systems

3 The challenge: ensuring and maintaining consistency

We assess the problem of ensuring and maintaining consistency between models and running system by considering difficulties in identifying relations between models and running system. The issues presented in this section are the result of literature study and previous experience in maintaining consistency of specific models. We discuss model *heterogeneity* in Sect. 3.1 (cf. Fig. 2), checking *alignment* of the running system with the models describing in Sect. 3.2 (cf. Fig. 2), and we analyze issues when *maintaining* such systems in Sect. 3.3.

3.1 Model heterogeneity

Different models have a different purpose and are, therefore, often denoted in different modeling languages. Checking consistency between such *heterogenous models* is a difficult process. We identify heterogeneity problems that make it hard to identify overlap and dependencies across models. We distinguish between *syntactic, semantic* and *pragmatic* heterogeneity.

3.1.1 Syntactic heterogeneity

When comparing models described in different languages, the first challenge is to look at the relation between the constructs of the languages. For example, an arrow in one language might be used to describe data flow, while in another one it denotes event flow. By comparing the syntax (i.e., the structure) of the different languages, relations and dependencies between them can be identified. These syntactic characteristics need to be identified by hand. The challenge is to identify both matching and non-matching concepts. Typically, conceptual modeling languages use concepts and relations between concepts to structure the world. Often, these concepts and relations appear to match ones in another language. However, usually there exist subtle differences between them, which need to be identified.

3.1.2 Semantic heterogeneity

Semantic heterogeneity is a broad research area closely related to ontology matching (e.g., [24,36,40]), where the challenge is to find a match between an ontology used in one model and the one used in another model. A common problem is to identify differently named concepts in two models referring to the same entity in the cooperation; i.e., to find coreferences within the different models. For example, one model might use the term "seller" where another one uses the term "provider". Another common problem is to identify homographs where one semantic concept is used in different models to refer to different entities in the cooperation. For example, in one model the term "seller" might refer to a wholesaler in the cooperation, while in another model it refers to the retailer that buys from the wholesaler. When modeling inter-organizational cooperations, semantic heterogeneity is often due to different model developers, different actors discussing the models, and different purposes of the models. Although ontology matching is a well-established research area, automatic ontology matching constitutes a challenge; i.e., many matchings are still done by hand which is a tedious process.

3.1.3 Pragmatic heterogeneity

We refer to heterogeneity between two conceptual models describing an inter-organizational cooperation not caused by semantical or syntactical differences as *pragmatic heterogeneity* (cf. [41]).

Perspective & focus. Regarding a particular inter-organizational cooperation, it is important to provide several models that together capture the full complexity of the cooperation. Every model *focusses* on one aspect of the system. In addition, the cooperation is described from different *perspectives*.

First, there is a choice how to *focus* the model. Typically, either *partial models* or *viewpoints* are used (cf. Fig. 3). In a partial model, the focus is on part of the cooperation that is described. For example, one model might describe interactions between all suppliers in a cooperation, while another one describes the interactions between one specific supplier and its customers. In a viewpoint, a particular aspect of the cooperation is modeled, ignoring all other aspects. For example, one model might focus on exchanged money, while another describes exchanged network messages.

Second, there is a choice from what *perspective* to describe the model. Typically, either a *single actor perspective* or a *bird's eye perspective* is taken (cf. Fig. 3). On the one hand, a model with single actor perspective ignores any information that is not related to this specific actor in the context of the considered cooperation. For example, a cooperation where a group of wholesalers sells goods to a group of retailers

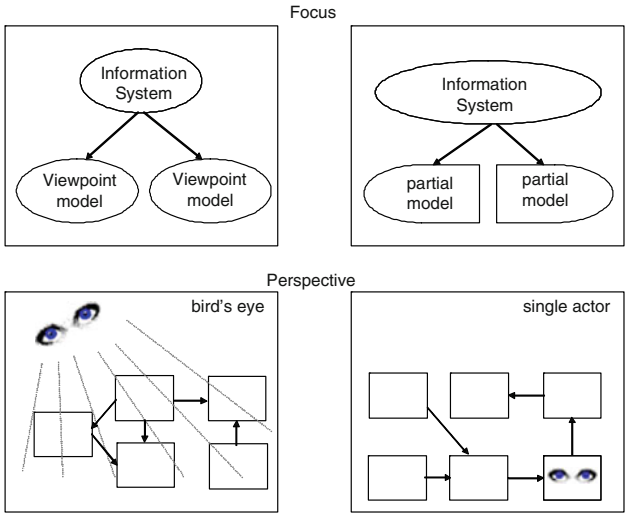

Fig. 3 Perspective and focus

might be described by a model depicting the relation between one specific retailer with wholesalers, ignoring retailers and wholesalers it has no relation with. On the other hand, a model from a bird's eye perspective describes the cooperation with all involved actors.

Both foci (i.e., partial models and viewpoints) can be described from both single actor perspective and bird's eye perspective. For example, a model might describe network connections in a cooperation (i.e., viewpoint) from the perspective of a specific seller (i.e., single actor perspective). Another model might describe just the suppliers (i.e., partial model) as well as all their relations (i.e., bird's eye view).

Granularity. Typically, a cooperation is described through models of different *granularity*. Granularity constitutes the level of detail with which the cooperation is described in a model. We distinguish between fine-grained models (i.e., more specific models) and *coarse-grained* ones. Coarsening a model, i.e., making it less detailed, filters out details not necessary for the purpose of the model [57]. We distinguish between *abstraction* and *generalization* [57] (cf. Fig. 4).

Coarsening through abstraction is the process of *leaving out* details on the cooperation (cf. Fig. 4) to reduce complexity. For example, a company might deliver its products using a transporting company, while a model describing this process might mention the transfer of products from company to customer, leaving out the transporter.

Coarsening through generalization is the process where *commonalities* between concepts or their relations are identified, and the result is used to describe a *set of* concepts or relations (cf. Fig. 4). In this case, no information is left out, but rather described on a higher level of detail. Common ways

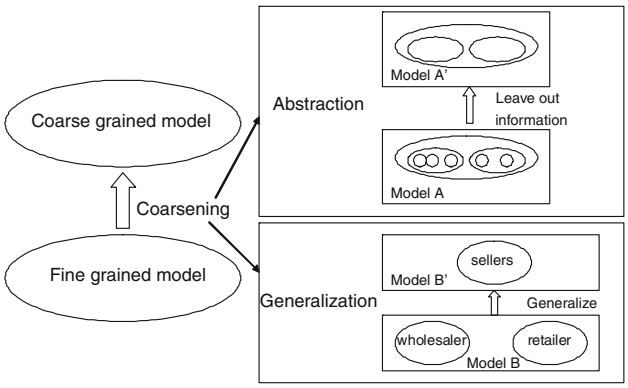

Fig. 4 Coarsening models

of generalizing in conceptual models for cooperations are (1) to identify *patterns* [33] and (2) to identify *hierarchies* [49].

The challenge at hand is to find relations and dependencies between concepts and relations in models of different granularity levels. For example, relating high-level concepts on sales targets in one model with low-level concepts on network exchanges in another model, is not straightforward. Another solution is to bring models to the same granular level by coarsening through abstraction and generalization. A consequence of coarsening models is loss of information.

Time frames. A third pragmatic heterogeneity factor concerns difference in *time frames* of models. Each conceptual model of a cooperation is meant for a specific period of time. The smallest possible time frame is captured in *instance-based* models, while other models describe a *period of time*.

The problem at hand is to check models for consistency while their time frames do not match. Consider the example where average commodity costs are determined per year and expected profit per month. It is difficult to ensure consistency since the current expected profit might not fit average commodity costs, while the remaining eleven months of profit might make up for this. Therefore, a choice in handling these time-frame differences needs to be made, and a first step is to recognize them.

Estimation and prescription. Since models considered in this paper describe cooperation as it should be, they are referred to as *prescriptive* models. Typically, these models describe *agreements* between different actors. For example, a model might describe that delivery of goods can only be done *after* having received payment. This behavior might be enforced in the implementation. However, besides agreements such models might also contain *estimations*. Typically, these estimations are done for that part of the cooperation which cannot be controlled through implementation, like customer behavior. Implementation of these estimations should *enable* estimated behavior as well as deviations from it.

Often, it is not obvious whether certain behavior is *estimated* or *agreed* upon. For example, a business model might describe an average of fifty customers per month (i.e., an estimated average) that should receive their ordered products on average in three days (i.e., an agreed average with the suppliers). Both averages are depicted as transfers between actors, leaving the difference between estimated and agreed average implicit. However, this difference should be implemented and when comparing high-level models (like business models) with more detailed, low-level ones that are directly implemented (like workflow models) this difference should become apparent.

3.2 Alignment with the running system

Aside from checking consistency between different models at design time, their consistency with the running system should be ensured as well (cf. Fig. 2). Checking a model against the running system is usually done based on *event logs* and is typically referred to as *conformance checking* [44] or *consistency checking* [3,19]. In particular, it is crucial to check whether models are implemented accurately, whether all actors behave according to the agreements made, and whether estimated behavior is indeed realized. An event log is consistent with a model if the essential parts of the model do not contradict reality, i.e., reality does not contradict the content of the event log, or vice versa.

In this paper, we assume that the event log is consistent with the running system (cf. Fig. 2). Consequently, the event log is used as correct representation of the running system. The first challenge is to identify which essential parts in the model actually *appear* in the event logs. For example, if estimations are done on the number of customers that register the coming month on a Web site, this data is detected as events in event logs. However, estimations on the male-female ratio of these registrations might not be visible in such log. The second challenge is to *abstract* essential information from event logs, i.e., to abstract information that enables consistency checking between running system and model. Typically, either the system is adapted in such a way that events entering the event log have the proper format or the necessary format is reconstructed from raw event logs after they are created. Although the first option, i.e., adaptation during runtime, is preferred since it is a one time effort, this is often not possible because of used software. As a consequence, event logs are often analyzed after runtime, i.e., necessary information is reconstructed.

3.3 Maintaining models

When checking consistency, contradictions between model and running system or between models might be detected. Ideally, such inconsistencies are resolved by adapting model

Table 1 Approaches for checking consistency

	Type of models		Type of consistency			Ensuring consistency			
	Viewpoints	Partial models	Inter-model consistency		Intra-model consistency	Checking			Construction
			Homogeneous	Heterogeneous		Testing	Translation		
							Overlap	Complete	
Mens et al. [32]		x		x	x	x			
Astesiano and Reggio [4]		x		x	x				x
Engels et al. [17]		x		x	x		x		
xlinkit: Nentwich et al. [34]		x		x				x	
Egyed and Medvidovic [16]		x		x					x
Varró and Pataricza [53]		x		x	x				x
χbel: Easterbrook and Chechik [14]	x		x					merge	
Uchitel and Chechik [52]	x		x					merge	
Fradet et al. [22]	x			x				x	
Bowman et al. [10,9]; Derrick et al. [12]	x			x				x	
Hunter and Nuseibeh [26]	x			x				x	
Viewpoints: Finkelstein et al. [20]	x			x				x	

or implementation. This is part of *model maintenance* which is particulary challenging since models overlap and contain dependencies. As a consequence, changing one model to regain consistency could introduce new inconsistencies with other models. Therefore, not only the inconsistency itself should be identified, but also its causes and, if possible, information about effects of changes in the model to regain consistency. This is typically not provided by multi-model consistency approaches where inconsistencies are identified, but no maintenance solutions are provided.

4 Related work

Terminology differs greatly among researchers. However, related work is done in terminology used in this paper. Table 1 provides an overview of the different approaches discussed. The table arranges approaches according to the *categorization* discussed in Sect. 2.3. The *type of models* that are handled by the approaches is specified in the first part of the table. In addition, the table shows whether approaches provide mechanisms to cope with *inter-model* and *intra-model* consistency constraints. Furthermore, some approaches are limited to *homogeneous* models, i.e., different models described in one language, while others are able to handle *heterogeneous models*. The last part of the table specifies how consistency is *ensured*. Some approaches provide mechanisms to *check* consistency through *testing, translation of overlapping parts*, or *complete translation* of models. Three approaches ensure consistency through *construction*.

Partial models. Mens et al. [32] target at consistent evolution of UML models. However, their approach also allows checking intra-model consistency. Their model checker implements the different UML metamodels that ensure intra-model consistency. Relations between metamodels ensure inter-model consistency. Since their approach implements existing metamodels (that specify consistency constraints), this approach is only usable in UML context.

Astesiano and Reggio [4] investigate existence of ambiguities and inconsistencies in UML language definition. In other words, the authors do not aim at solving inconsistencies between UML diagrams for a particular specification. Their goal is to develop a consistent UML language. They reduce inconsistencies by improving the metamodels and they rely on translating models. Therefore, it is classified as achieving consistency through construction by improving the metamodel (cf. Table 1).

Engels et al. [17] develop a method for checking consistency of UML models to decide at which point in time of the development process UML partial models should be consistent with each other. In UML, consistency requirements exist that specify consistency relations between different model types (e.g., a statechart has to accept each stimulus a sequence diagram specifies). Their implementation (the Consistency Workbench) tests whether two models are consistent against these consistency rules. They formalize overlapping parts of the models into a *common semantic domain*. A discovered inconsistency is either tolerated or resolved. Their approach suits horizontal and evolution consistency [18]. With evolution consistency the emphasis

lies on preserving model aspects while it is evolving. This is achieved by adding implementation rules. This approach uses several semantic domains for one set of models. Therefore, relations between constraints across domains are not expressed. Our method avoids using different semantic domains to avoid losing these relations.

xlinkit [34] is a method for expressing constraints across heterogeneous models. It offers semantics that shows links between two mutually inconsistent elements of different models. Focus is on identifying inconsistencies rather than solving them. Nentwich et al. [35] extend this diagnostic method with a repair actions method. Although xlinkit is mainly used for UML models the authors argue that their method is language-independent. Consistency rules are expressed using a restricted form of first order logic. Furthermore, models are transformed in XML. These restrictions make xlinkit unsuitable for our problem.

Egyed and Medvidovic [16] provide an approach for heterogeneous software development. It enables refining an architectural design model into UML models. It ensures initial consistency since it is a unidirectional approach. Therefore, any updates to UML models or refinement of UML models into other UML models might interfere with the original architectural model. To overcome the problem of further refinement of UML models and their possible inconsistency with the overall architectural model, abstractions from concrete models to abstract ones (vertical) and from specific ones to generic ones (e.g., instance to class) are supported. The authors state the approach is language independent, but it is only illustrated by transforming C2 models into UML.

Many approaches rely on model transformations using an intermediate universal language (e.g. XML). A correctly defined metamodel is crucial to handle such transformations. [53] tackle several metamodeling problems (causing inconsistencies) by defining rules in their construction. Their method which is also applicable to UML. Their multilevel metamodeling approach overcomes the well-known problem of concept replication. Heterogeneous refinement is supported. Although focus is on metamodeling, it facilitates consistent model development.

Viewpoints. Easterbrook and Chechik [14] develop the χbel framework for merging state machine models that describe different behavior. χcheck is a model checker which analyzes properties of merged models. Multi-valued logics is used which supports statements like "the majority says" instead of "true" and "false" (i.e., "everyone says" and "no one says"). This enables reasoning over inconsistent models so that stakeholders can discuss different options when merging models. The merging and reasoning process depends on relations models have with each other. The approach is suitable for homogeneous models and is, therefore, not applicable to our problem.

Uchitel and Chechik [52] develop an approach for merging partial behavioral models in the same language. Here, *partial* refers to partial behavior where models provide concepts for not yet known behavior. Their approach is suitable for viewpoint models. It assumes models being intra-model consistent, and argues that the result of the merger should be the minimal common refinement of considered models. The result is said to be consistent if it is a common refinement of both models. This approach is suitable for any state-based behavioral system with formal semantics. It focuses on observable behavior of models rather than on structural aspects. Restrictions regarding homogeneous state-based models are not sufficient in the context of our problem statement.

Fradet et al. [22] develop a method for defining multiple view architectures. Their approach is formal, suitable for heterogeneous models, and not language-specific. The structural part of each view is represented as uninterpreted graph. Consistency between different views is checked on graphs by means of an algorithm. The interpretation of graphs is done through formulating a set of constraints that specifies both intra-model and inter-model requirements. Although it is recognized that formalizing all constraints in graphs might be cumbersome for large models, the authors state that this is not a problem since their graphs can be expressed in terms of constraints. Their approach is specifically designed for software architectures where focus is on different components and their connections. More specifically, they focus on the structural part of the architecture, while our method aims at a more complete approach where also communication between different actors is modeled.

Consistency checking is a big concern in Open Distributed Processing (ODP). ODP provides a method for distributed development where five viewpoints are proposed for the modeling phase: enterprise, information, computational, engineering, and technology. The requirements modeled in each of these viewpoints should be reflected in the overall description of the system. Approaches to ensure or check existence of such a description are proposed by [9,10] and [12]. They aim at checking and ensuring existence of an overall system description capturing each viewpoint. Consistency between specifications is defined as the existence of an internally valid description which represents all requirements of the specifications. Consistency is checked between a viewpoint specification and the overall description, but not between two viewpoints. As a result, consistency requirements between one viewpoint and the overall description might differ from requirements between another viewpoint and the overall description. This difference occurs when viewpoints are described in different languages and have different levels of abstractions. The authors refer to this as unbalanced consistency. By doing bilateral consistency checks, it is difficult to create global consistency: Each viewpoint might

be consistent with one or more descriptions, but finding a description which is consistent with all viewpoints is hard. Therefore, they propose to ensure global consistency by unification of viewpoints. Two viewpoints are unified after one of them is translated into the other, or after both of them are represented in a common semantic domain. By showing intra-model consistency of the unified model, two viewpoints are proven to be consistent with each other. Now, a third viewpoint can be unified in the same manner with the result of the previous unification. For our purpose, this approach is less suitable since it relies on a universal language for all viewpoints.

Hunter and Nuseibeh [26] propose a logic-based approach for reasoning in the presence of inconsistencies. They propose quasi-classical logic, which is a weakened version of classical logic. It allows reasoning with inconsistencies by deriving all resolvants of assumptions without allowing trivial derivations. This approach first labels information and then these labels are propagated through the reasoning process. This allows tracking inconsistent information and provides a better problem analysis. Furthermore, this approach computes maximally consistent subsets of the inconsistent information. Obviously, this means that viewpoints are translated into logic.

The Viewpoints framework, suggested by Finkelstein et al. [20], and its extensions by Easterbrook et al. [15] and Finkelstein et al. [21] allow inconsistencies when developing different viewpoints in order not to kill creativity. Their method is logic-based and allows checking intra-model and inter-model consistency. The viewpoints are translated into logic and tested against predefined consistency and well-formedness rules. Consistency rules are created per viewpoint and are locally stored. So, there is no overall consistency checking mechanism. The authors assume that each developer is concerned with consistency of the viewpoint he is working on. If a rule is violated, its "owner" determines whether or not it should be resolved. Focus is rather on bilateral consistency, resulting in a local view. As a result, it is not calculated what the effects are on relations other than the bilateral relation that is dealt with.

Overview approaches. Spanoudakis and Zisman [48] introduce a six step approach to maintain consistency in model development. This is based on unification of several other approaches and tries to cover all concerns. The six steps are as follows: detecting overlap, detecting inconsistencies, diagnosis of inconsistencies, handling inconsistencies, tracking, and specification and application of an inconsistency management policy. An interesting detail in this method is the distinction between different overlap types. Many approaches assume that there either exists a complete overlap or no overlap at all, which is not realistic. Detection of inconsistencies is done with logic-based approaches, model checking approaches, specialized model analysis, or with human centered exploration. The authors state that the main challenges in inconsistency detection are efficiency and scalability, especially when dealing with evolving models. Although this approach does not provide a tool or specific approach for maintaining consistency, the identification of concerns in maintaining consistency relations is impressive.

Nuseibeh et al. [37] describe a method for managing consistency in software development. Consistency rules detect inconsistencies during monitoring. The inconsistency is analyzed, and a solution provided. Consequences of handling inconsistencies are, in turn, monitored. They emphasize it is difficult to decide *when* to check consistency. During model development, inconsistencies might be tolerated. Many of them sort out themselves. Obviously, hoping that an inconsistency resolves itself or does not cause any problems, causes significant risks in information systems engineering.

In conclusion, there exist many approaches for checking or maintaining consistency over many different disciplines. Many solutions provide a high-level description of the problem and lack suggestions how to solve it. Other solutions are often language-specific and, therefore, not usable in other domains. The few approaches that are on the right level of abstraction either lack the option to apply them in a heterogeneous context or rely on a common semantic domain (i.e., an universal language is assumed).

5 Our MaDe4IC method

We discuss our MaDe4IC method for ensuring and maintaining consistency within and between inter-organizational models. The presented approach is derived from literature study and previous experience in maintaining consistency of specific models. We start with discussing model characteristics in Sect. 5.1, after which we give an overview of the different steps in Sect. 5.2. In Sect. 5.3, we discuss the running example. We conclude with a detailed description of each step of our method in Sects. 5.4–5.8.

5.1 Model characteristics

A typical conceptual model consists of *concepts* representing *entities* from the real world. These entities are characterized by some *properties* which are also represented by the concepts. Furthermore, *relations* are used to depict interrelated concepts. Typically, models of cooperative systems abstract from any internal behavior and focus on exchanging objects between actors. Since these exchanges are an essential part of the cooperative system, they are likely to be included in several models on comparable granularity. In the following, the models of cooperative systems contain the following types of concepts and relations:

1. Actors in the cooperation (i.e., parties cooperating together).
2. Exchanged objects (such as information and money) that are used to establish the relation between actors.
3. Relations between concepts, i.e., between actors and between exchanged objects in different models.

For example, a concept representing a bike might have the property "price", indicating the specified bike has a certain price. Furthermore, the *nature* of the relation might differ. For example, there are causal and temporal relations indicating some concepts have a causal relationship, while others have a temporal one.

Which real-life entities and relations in the cooperation are captured by a specific model depends on the type of model used. Our method is applicable to cooperative systems showing these characteristics. It focuses on identifying dependencies between concepts within and between models. These dependencies form the basis for managing the models by analyzing causes for inconsistencies.

5.2 Our method: an overview

Figure 5 gives an overview of our approach. It is divided in five phases. Input is provided by different conceptual models developed for the inter-organizational cooperation.

In the first phase, **model analysis**, preparations are done for identifying dependencies within and between models. This phase comprises two steps. In **Step 1**, each model is analyzed resulting in model characteristics. The latter are used to homogenize models in **Step 2** to make these (potentially heterogeneous) models comparable. We give some guidelines to tackle heterogeneity problems in a structured way, without providing details on how to do this.

In the second phase, **inter-model analysis**, we identify different model relations and use them to define consistency constraints. This phase also comprises two steps. In **Step 3**, we compare each pair of models, using knowledge about their characteristics (cf. Step 2) to identify inter-model relations. For example, the existence of a concept in one model might depend on the existence of a concept in another model. Based on these relations, for each model pair we define a set of consistency constraints in **Step 4**. These constraints explicate when we consider two models being consistent with each other. For example, if the existence of a concept from one model depends on the existence of a concept from another model, the constraint explicates this dependency relation. Models are considered being inconsistent if a concept exists in one model, but not in the other.

In the third phase, **intra-model analysis**, for each model we identify in **Step 5** the type of relations used to connect

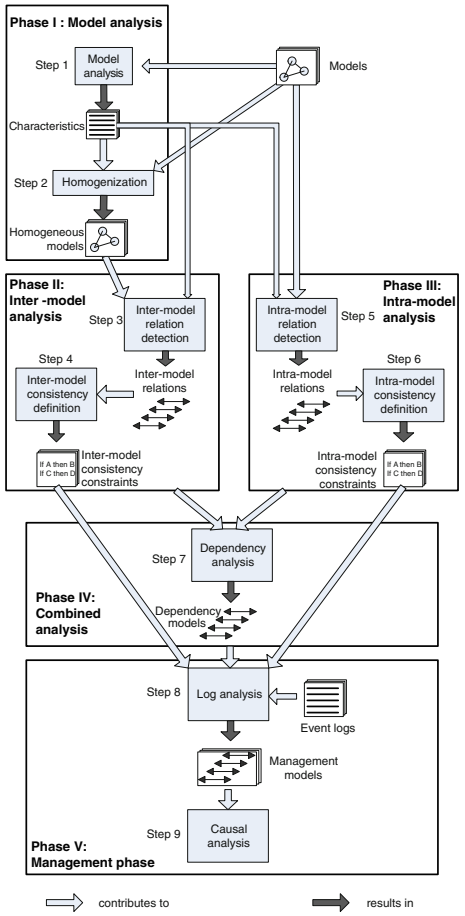

Fig. 5 Managing dependency relations in inter-organizational models

the different concepts. This analysis helps us to explicate intra-model consistency constraints in **Step 6**. These constraints describe when a model is considered being consistent with the running system.

In the fourth phase, **combined analysis**, the identified dependency relations and the consistency constraints are used to perform a combined dependency analysis in **Step 7** that creates the combined dependency models. Together, these models describe the different dependencies and consistency constraints in a structured way so that they form a proper base for managing the models. These dependency models are formal models which enables easy implementation. Furthermore, they only depict those parts of the original models that influence the consistency constraints.

The fifth phase, **management phase**, occurs at runtime of the system and comprises two steps. **Step 8** checks consistency constraints in the *log analysis* against the event logs where the dependency models are used to construct feedback for managing the models. This feedback enables causal analysis for constraint violation as well as prediction of consequences for solving inconsistencies in **Step 9**.

The first seven steps are done manually and (possibly) before the system is running. Steps 8 and 9, the actual

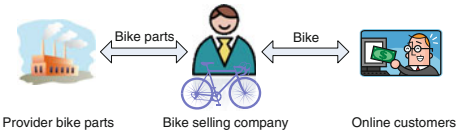

Fig. 6 Running example: selling bikes

management steps, can be automated through implementation, using the formalization created in Step 7.

5.3 Running example

We introduce a running example to illustrate the different steps of our method (cf. Fig. 6). In this example, we are interested in the cooperation between a company selling bikes online, and its customers and providers. The company offers both *mountain bikes* and *city bikes* for online purchase. The company buys the bikes in parts from a provider, then assembles them and finally resells them. Each product is sent to the customer for a fixed delivery price. In addition, a customer can choose fast delivery, or delivery on a specific moment like evenings or weekends. For such service an additional fee is calculated.

5.4 Phase I: model analysis

In the model analysis part of our method, first of all, each model is analyzed to identify its specific characteristics (Step 1). Following this, related model pairs are homogenized (Step 2). Homogenization is not the core part of this paper. However, in consistency checking, it is inevitable to address this problem. Therefore, we provide a set of guidelines in Step 2 to stepwise tackle the problem without discussing solutions in detail. In the remainder of this paper we assume heterogeneity problems are overcome.

5.4.1 Step 1: model analysis

> Goal: Identify model characteristics of each conceptual model of the cooperation in order to enable inter-model and intra-model analysis.

Our method starts with an characteristics analysis of each model (cf. Step 1 in Fig. 5). We identify the different conceptual characteristics as discussed in Sect. 3.1.3. These characteristics are later on used to enable inter-model as well as intra-model analysis. We analyze models for the following characteristics:

(i) Focus:	Viewpoint	↔	Partial model
(ii) Perspective:	Single actor	↔	Bird's eye view
(iii) Property type:	Estimation	↔	Prescription
(iv) Time frame:	Instance-based	↔	Period of time

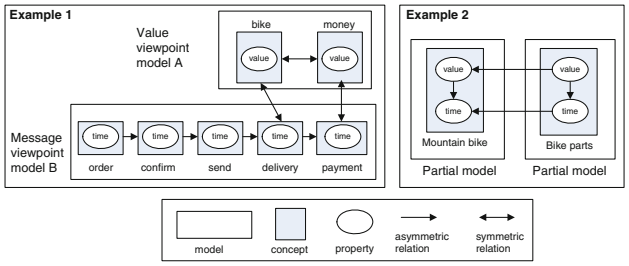

Fig. 7 Dependency relations between viewpoints and partial models

To illustrate Step 1, we consider two examples:

Example 1 When considering our running example described in Sect. 5.3 a possible choice is to develop two viewpoint models. Assume model A describes exchanges that have some value between customer and company, while model B describes messages exchanged between actors. The left part of Fig. 7 depicts these two models.

Example 2 Another possible choice for modeling our running example is to develop two partial models. The right side of Fig. 7 depicts two such models where each one describes one entity with different properties. The *mountain bike* is the product purchased by the customer, while the *bike parts* are represented by the box containing the unassembled mountain bike as purchased by the company.

(i) Focus: A model focusses on a subset of entities as well as relations of the system (i.e., a partial model), or on the complete system with focus on a specific characteristic (i.e., a viewpoint model). Typically, models related to the same system have the same focus type. Example 1 in the left part of Fig. 7 depicts our scenario with two viewpoint models, while Example 2 in the right part of the figure depicts two partial models. Identifying the focus type constitutes an important prerequisite for both intra-model and inter-model analysis (cf. Fig. 5).

(ii) Perspective: A model can be described from a local perspective (i.e., single actor perspective) or from a global one where the system is considered as a whole (i.e., a bird's eye perspective). The models in Fig. 7 all reflect a single actor perspective, i.e., they are described from the perspective of the company. Other than with focus, perspectives often differ between models describing one system. In such situation, it is important to identify how the part description from the single actor perspective is related to the whole description of the bird's eye perspective. This heterogeneity difference is addressed rather than homogenized since perspective choice is model-specific and should not be changed.

(iii) Property type: The property type describes whether a property value is an estimation or prescription. This can differ *within* a model where some properties are estimations, while others are prescriptions. Consider, for example, the left part of Fig. 7 where the property values which *prescribe* behavior.

The message viewpoint, however, describes an expected time frame in which messages are exchanged. There are estimations on exchange times described with the property 'time'. The estimations are *predictions* of reality, rather than prescriptions.

(iv) Time frame: The time frame used in the model should be identified. Especially whether a model is valid for a specific period of time, or whether it is an instance-based model. For example, a model might describe behavior for the coming half year (period of time) or describe behavior for each invocation separately (instance-based). This information is needed to homogenize the models for the inter-model analysis (cf. Fig. 5).

The results of Step 1 are as follows:

- Each model is characterized by its *focus*, *perspective*, and *time frame*.
- Each property in the model is characterized by its *property type*.

5.4.2 Step 2: homogenization

Goal: Identify heterogeneity between model pairs using the model characteristics identified in Step 1. Homogenize differences between model pairs if possible.

We *homogenize* the different models to enable their comparison (cf. Step 2 in Fig. 5). For homogenization we need to overcome differences between the models. For this, we address *syntactic, semantic* and *pragmatic* heterogeneity (cf. Sect. 3.1). Certain heterogeneity problems are addressed in later steps, while others are homogenized in Step 2:

- syntactic: addressed in later steps
- semantic:
  - coreferences: homogenize in Step 2
  - homographs: homogenize in Step 2
- pragmatic:
  - perspective: addressed in later steps
  - focus: addressed in later steps
  - granularity: homogenize in Step 2
  - time frame: homogenize in Step 2
  - estimation and prescription: addressed in later steps

Syntactic heterogeneity is inherently present since models are simply described using different languages. The key is to identify syntactic commonalities and differences across the modeling languages. Explicating differences and commonalities between *concepts*, *relations*, and *properties* of two models suffices.

Example The models from Fig. 7 are described in the same language, i.e., they use the same constructs for concepts, relations, and properties. There are no syntactic differences.

Furthermore, *semantic heterogeneity* must be addressed, especially identifying and solving coreferences and homographs (cf. Sect. 3.1.2):

Example The value viewpoint model A in Fig. 7 describes a concept *bike*, while the partial model in the same figure models a concept *mountain bike*. Although the concepts differ semantically, they refer to the same real-life entity, i.e., they are coreferences.

This needs to be identified and resolved by choosing one semantics to describe the entity. Also if two semantically equal concepts in different models refer to different entities (i.e., homographs), one concept should be renamed.

In addition, some *pragmatic heterogeneities* can be homogenized in the current step, while others are simply recognized here, and addressed when comparing the models in later steps. Addressing heterogeneity later means it then becomes more difficult to find overlap or relations between models since they are still heterogeneous. However, by identifying where the models are different in the current step, relating the models in later steps becomes easier. Concerning pragmatic homogenization, only *time frame* differences and *granularity* differences are homogenized.

A *time frame* difference between two models needs to be resolved before models can be compared in the following steps. Changing the time frame of a model such that it fits the time frame of another model is not a straightforward process. It requires discussions with all involved actors to come to an agreement on how to handle the changes. Generally, there are two approaches that can be followed: either a time frame is shortened or it is lengthened.

Example The Value viewpoint model A from Fig. 7 describes the monetary relation between *bikes* the company sells and *money* the company receives for this. The model might be valid for a period of one year. The related Message viewpoint model B, however, describes necessary message exchanges between partners per bike sale, i.e., the time frame is instance-based. This time frame difference is resolved by shortening the time frame of the Value viewpoint model A from a period of one year to an instance-based model.

When *shortening the time frame*, challenges arise with agreed upon *average* values and with *indivisible* exchanges. For example, if a contract specifies an average value to accomplish over twelve months, this value cannot be assumed to hold in the first six months only (e.g., due to seasonal behavior). Furthermore, if there are exchanges between

actors that only occur once during a period (e.g., selling one bike), shortening this period implies dividing an atomic exchange which is not always possible.

When *extending a time frame*, in turn, the biggest challenge is to estimate content of future contracts and models. For example, if a contract is specified for the coming year and we need to extend the model for the coming two years, either all contracts need to be extended or estimations on future contracts become necessary. Therefore, the changes typically come with estimations of behavior.

The second pragmatic difference between models constitutes *granularity* difference between models. To check consistency between two models, they need to describe the system on the same level of granularity. Coarsening is typically used to overcome granularity differences as described in Sect. 3.1.3. Whether abstraction or generalization techniques are used needs to be decided by the developer and will differ per model. Coarsening is done *per model pair* since it is important to consider models on the most fine-grained level as possible.

Results of Step 2 are as follows:

- *All models* are semantically homogeneous,
- *Each model pair* is time frame homogeneous, and
- *Each model pair* has the same level of granularity.

5.5 Phase II: inter-model analysis

The goal of inter-model analysis is to identify dependencies *between* different models describing the same system, and to use these dependencies for explicating inter-model consistency constraints. These constraints are then checked and ensured during the management phase (cf. Fig. 5). We first identify inter-model relations between homogenized models (Step 3). Then we define inter-model consistency constraints based on identified relations (Step 4).

5.5.1 Step 3: inter-model relation detection

> Goal: Identify inter-model relations for each model pair. More specifically, we identify *dependencies* between models, concepts, and properties.

When considering partial models, models describing different system parts are interconnected. This interconnection needs to be identified. For example, if one partial model describes interactions on the customer side, and another one describes interactions on the provider side, these two models need to be interconnected before implementation. When considering viewpoint models, *overlapping* parts (i.e., where two models describe the same system) need to be identified. For example, if one viewpoint model describes which monetary actions are expected in a cooperation, and another one describes the order in which messages are exchanged, the monetary actions and messages that refer to the same real-life entity need to be related. These inter-model relations are used in Step 4 to formulate consistency constraints.

The term *relationship* has a broad meaning. Therefore, we start with explicating which types of relations are distinguished. The goal is to ensure consistency between models which is accomplished by identifying those parts of the models that influence each other. We consider any type of *dependency* relation between concepts. This dependency can be strong as, for example, in a causal relation where one concept causes another one to occur. However, also less strong dependencies are considered, e.g., in a temporal dependency one concept always occurs before another one without being its cause for occurrence. Further, relations can be *symmetric*:

Example In Fig. 7, *payment* of a bike (Value viewpoint model) and its *delivery* (Message viewpoint model) have a symmetric relation since there is no payment without delivery and no delivery without payment.

Relations can be *asymmetric* as well. For example, regarding *payment* of a bike and its *bill*, the existence of the payment depends on the existence of a bill, whereas the bill can also exist without the existence of a payment.

We distinguish relations between *concepts* and those between *properties*. A dependency relation between concepts is referred to as *existence dependency*, while a dependency relation between properties is called *property dependency*.

Existence and property dependencies. An *existence dependency* between concepts indicates that *existence* of these concepts depends on each other.

Example Considering our running example, the existence of a concept referring to the payment of a mountain bike by a customer depends on the existence of a concept referring to the actual bill created by the company stating the total cost of the purchase. In other words, without such a bill the customer will not pay.

A *property dependency* indicates the *value* of one property depends on the value of the other one:

Example The amount of money (represented as property *value* of concept *mountain bike*, cf. Fig. 7) paid by the customer for a bike depends on the amount of money (represented as property *value* of concept *bike parts*) paid by the company to its suppliers for the bike parts. Here, the value property of one concept depends on the one of another concept.

If properties of two concepts depend on each other, the concepts themselves have an existence dependency :

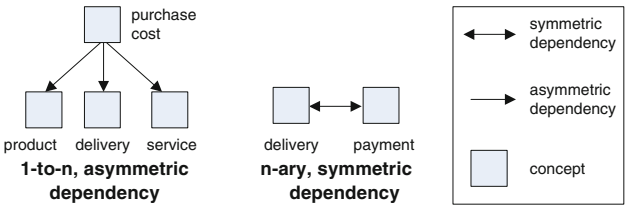

Fig. 8 Example: asymmetric and symmetric dependency relations

Example Since the properties of concepts mountain bike and bike parts depend on each other (cf. Fig. 7), the two concepts are existence dependent on each other. The existence of concept *mountain bike* depends on the existence of concept *bike parts* since there exists no bike if there are no bike parts.

In other words, if two values depend on each other, existence of concepts containing these values also depends on each other.

Symmetric and asymmetric dependencies. A second dependency characteristic is its *direction*. A dependency can be *asymmetric* where one concept depends on one or more other concepts:

Example A cost dependency model depicts where costs of purchasing a bike come from (cf. left part of Fig. 8). *Purchase cost* depend asymmetrically on the purchased *product*, *delivery* cost, and additional *service* costs for fast or weekend delivery.

Alternatively, a dependency relation can be *symmetric* where concepts depend on each other:

Example A dependency model (cf. right part of Fig. 8) depicts a dependency between delivering the bike by the company, and payment of the costs by the customer. Delivery depends on payment, and vice versa since it is not specified whether payment occurs before or after delivery, i.e., they have a symmetric dependency.

In summary, we distinguish the following four types of dependency relations:

- Asymmetric existence dependency, where the existence of a concept depends on the existence of one or several other concepts,
- Asymmetric property dependency, where the property value depends on one or several other property values,
- Symmetric existence dependency, where the existence of concepts depend on each other,
- Symmetric property dependency, where property values depend on each other.

How to identify inter-model relations is explained next, where these relations depict for viewpoint models *overlapping* parts and for partial models *connecting* parts.

Viewpoints. When identifying relations between two viewpoint models, the goal is to identify their *overlap*. We adopt the definition of overlap as given by [17]:

> "Overlap is defined as two specifications which are not independent since they describe common aspects."

Our aim is to find these common aspects, and to identify the type of relation they have. In conceptual modeling, the common aspects are concepts with potentially different properties referring to the same real-life entity. Preprocessing models in the homogenization phase (Step 2) eases identification of concepts referring to the same entity. Identifying the relation type between these commonalities is done by hand:

Example The left part of Fig. 7 depicts a Value and Message viewpoint model with their inter-model dependencies. We assume model engineers refer to a bike sold to customers with concepts *bike* in model A and *delivery* in model B. Further, they refer to money paid by the customer with concepts *money* in model A and *payment* in model B. In both cases, concepts refer to the same entity in the real world, and existence of one concept assumes existence of the other. For example, if a bike is delivered (Message viewpoint model), it is also paid for (Value viewpoint model). Therefore, they have a symmetric existence dependency.

Partial models. When identifying relations between partial models, the goal is to identify relations between concepts, referring to different real-life entities, and between the same properties in different concepts. This can be a more challenging task than identifying overlap in viewpoints since there are no commonalities to observe; each entity is captured in only one model:

Example In the right part of Fig. 7, one partial model depicts the *mountain bike* and the other one the *bike parts*. The models refer to different entities, but there exist dependencies between them. For example, the bike value depends on its parts value. Identifying such dependencies constitutes a challenge since there is no overlap between the models.

Usually, there exists some (implicit) model that links partial models. For example, if a cooperation is developed, there exists a common model that explains which partial model is connected to which other one, and how this connection looks like. However, these are high level connections, describing the relation on model level rather than on concept level:

Example In the right part of Fig. 7, the partial models are related. There exists a common model describing that the mountain bike depends on its parts (i.e., a common model describing high-level connections). The challenge, however,

is to find the relations between these two models on a concept level. For example, what is the relation between delivery time of bike parts and possible delivery time of the bike.

With our method we identify different properties that are modeled for each real-life entity in a concept. This way, property dependencies are identified between the same properties in different, but related concepts. In partial models, each real-life entity is represented in one model as a concept with some properties. For example, a concept in a partial model might contain information on its monetary value, size, and validity. Concepts in related models are related through identifying equal properties. For example, the monetary value of one concept might depend on the monetary value of other concepts, while their physical size is unrelated:

Example In the right part of Fig. 7, a schematic representation identifying these relations is represented. Each partial model describes one entity with different properties. The value of the mountain bike depends on the one of its parts. The possible delivery time depends on the delivery time of its parts. These are asymmetric property dependency relations.

The results of this step are as follows:

- Identification of inter-model dependency relations between concepts and properties,
- Inter-model relations are categorized as follows (i) symmetric and asymmetric, and (ii) property and existence dependencies.

5.5.2 Step 4: inter-model consistency constraints

Goal: Identify inter-model consistency constraints for each model pair. More specifically, we use the identified dependency relations from Step 3 to formulate constraints for each relation.

We consider two models being consistent if they have no *contradictions*. Since contradictions only occur in *related* models, they only emerge in *inter-dependent* model parts. In Step 3, we identify these inter-model dependencies. In Step 4, we now use them to formulate *consistency constraints*. These constraints are divided into different categories, in analogy to the different dependency relations introduced in Step 3. For each identified dependency, we formulate a consistency constraint. Each dependency constraint is formulated according to the related dependency type:

- If concept x is **asymmetric existence dependent** on set Y of one or more concepts, the constraint states: If x exists then Y exists.
- If concepts in set X of two or more concepts are **symmetric existence dependent** on each other, the corresponding constraint states: If $x \in X$ exists, all concepts in X exist.

Example In the left part of Fig. 7, the viewpoint models have a symmetric existence dependency relation as discussed in Step 3. These relations are translated into consistency constraints. For example, concepts *bike* of Model A and *delivery* of Model B are symmetric existence dependent. The corresponding constraint states: If concept *bike* exists, concept *delivery exists*, and vice versa. This constraint expresses that if at runtime the bike gets delivered, it is also paid for, and vice versa. If this is not the case, the constraint is violated.

In case of asymmetric and symmetric property dependencies, there exists a relation between property *values*. The exact relation between the values is model dependent. For example, property value of x might be *twice as big* as related property value of y. The general format for defining consistency constraints for asymmetric and symmetric property dependency is as follows:

- If concept x is **asymmetric property dependent** on set Y of one or more concepts, and z is the predicate describing this relation, the corresponding constraint states: Property value of x relates to property values of Y according to predicate z.
- If concepts in set X of two or more concepts are **symmetric property dependent** on each other, and z is the predicate describing this relation, the corresponding constraint states: For each $x \in X$ it holds that property value of x relates to all other concepts in X according to predicate z.

Example In the right part of Fig. 7, the partial models have asymmetric property dependency relations as discussed in Step 3. Assume that the relation between value property of *Mountain bike* and value property of *Bike parts* states the value property of *Mountain bike* is always larger than the one of *Bike parts*. Then, this relation is translated into the following consistency constraint: Concept *Mountain bike* is asymmetric property dependent on concept *Bike parts*, where the property value of *Mountain bike* is larger than the one of *Bike parts* (i.e., this is the predicate). The corresponding constraint states: The property value of *Mountain bike* is larger than the one of *Bike parts*. This constraint depicts at runtime the value of the bike is larger than the value of its parts. If this is not the case, the constraint is violated.

Often it is possible to formulate *generalized* consistency constraints where constraints are defined over a *set* of inter-model dependencies. Since every identified dependency relation results in a consistency constraint, it is convenient to explicate one general consistency constraint that captures a set of similar single constraints.

Example If hundreds of payments in one model are existence dependent on a bill in another model, each of these

dependency relations results in a separate constraint stating "If payment A exists, also bill B exists". However, a single constraint stating: "If a payment with characteristic Y exists, also a bill with characteristic Z exists", would generalize over this large space of constraints.

Generalization is applicable in both viewpoint models and partial models. For both it holds that generalization is only possible over related combinations *with the same type of dependency relation* and, therefore, over consistency constraints of the same type. For example, it is not possible to define a general consistency constraint for two sets of related concepts where one set has an asymmetric existence relation and the other one a symmetric existence relation. Generalization constraints differ for viewpoint models and partial models. Therefore, we discuss them separately:

Viewpoint models. In viewpoint models, often a set of concepts in one model is related to concepts in another model. Currently, each of these relations has a separate constraint:

Example In the left part of Fig. 7, two symmetric existence dependency relations are depicted. Fig. 9 shows these two relations. The value viewpoint model describes which products are transferred. The message viewpoint model, in turn, specifies which messages are exchanged. If a message indicates the bike is delivered or a payment is done (message viewpoint), it is concluded the bike or payment is received (value viewpoint), respectively. If the bike or money has exchanged hands (value viewpoint), it further is concluded there must be a message confirming bike delivery or money payment (message viewpoint). Therefore, existence of a good transfer (e.g., bike) and existence of a message exchange (e.g., delivery message) have a symmetric dependency. Without generalization, there are two constraints describing the relation between value and message viewpoint model. One states that if the bike is transferred also a message confirming the bike delivery exists and vice versa. The second one states that if money for the bike is transferred, a message confirming the bike payment exists and vice versa. We construct a general consistency constraint that covers both constraints: "If a valuable product is handed over, a message confirming this exists and vice versa".

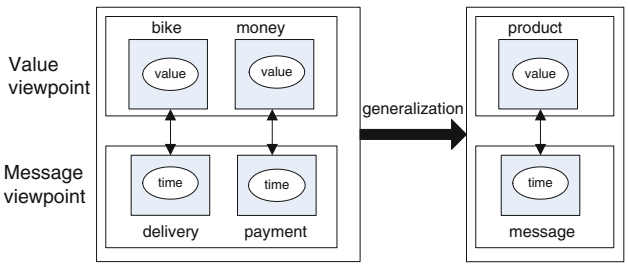

Fig. 9 Possible generalization over symmetric consistency constraints

Generalization in viewpoint models is possible if there are inter-model consistency constraints based on the same type of relation (e.g., symmetric existence dependency). If generalization of property dependency constraints occurs, properties in the different constraints are the same. For example, if one constraint describes a dependency between two time frames, while another one describes a dependency between monetary values, it is not possible to generalize over these two constraints.

This general constraint typically describes the *nature* of the models. Here, we state that every concept that represents some valuable transfer in the value viewpoint has a related concept in the message viewpoint describing the message transfer:

Example In a value and message viewpoint, several entities in the cooperation are modeled in both viewpoints. For example, in the left part of Fig. 7, money paid for a bike is modeled as valuable transfer in the first viewpoint, and as message transfer in the second one. However, not every concept describing a message transfer has a monetary value.

Therefore, we state that for each message transfer in the message viewpoint containing some value there is a concept representing this value in the value viewpoint model. With this consistency constraint we check whether all necessary concepts are modeled or whether a model is incomplete.

Aside from checking whether all concepts are present, it is important for each business transaction to check whether both models describe the same *set* of transfers. For example, for viewpoints model entities "bike" and "money" for the bike, it is important that no model assumes *two* bike transfers occur for one money transfer, while the other model assumes *one* bike transfer occurs for a money transfer. Therefore, an additional general consistency constraint describes that for each business transaction modeled in the viewpoint models, the concepts should occur in the same setting.

Partial models. In partial models, we often observe that property values of a set of concepts in different models are dependent on each other:

Example In the left part of Fig. 10, the concept *customer payment* has three dependencies; money paid by the customer depends on special costs but also on product costs and delivery costs. Each property dependency relation has a separate constraint clarifying how customer payment depends on the different costs. For example, one constraint might state: "The amount of money paid by the customer contains special costs".

However, it is possible to construct one general constraint that describes the three separate ones:

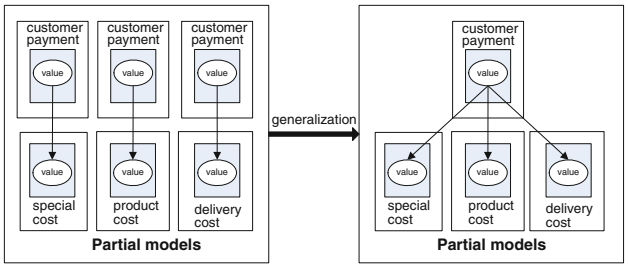

Fig. 10 Generalization over asymmetric consistency constraints

Example The right part of Fig. 10 depicts that *customer payment* and its three relations are interconnected. Now, the general constraint states: "The amount of money paid by the customer is at least the sum of special costs, product costs, and delivery costs."

In general, we state that constructing a generalization of constraints in partial models is possible if (1) the dependency relations described by the constraints are interconnected, and (2) they connect the same type of property. For asymmetric relations, it should be possible to form a tree out of the dependency relations and for symmetric ones it should be possible to create an undirected graph from the dependency relations.

Results of Step 4 are as follows:

- Identification of consistency constraints for both symmetric and asymmetric dependencies, and property and existence dependencies, and
- Illustrating generalization of consistency constraints.

5.6 Phase III: intra-model analysis

Intra-model analysis identifies dependencies between different concepts *within* one model (Step 5). Further, we define intra-model consistency constraints based on these relations (Step 6).

5.6.1 Step 5: intra-model relation detection

Goal: Identify intra-model relations for each model. More specifically, we identify *dependencies* between concepts and properties.

For detecting intra-model relations, we use the original models and not the homogenized ones since we want to preserve as much original data as possible. Although conceptual models all describe *relations* between concepts and properties, the relation *type* differs. Therefore, we explicate the relation type used in the language for depicting relations between concepts and properties within a single model. More than one type of relationship can be used in a model. We use the same distinction as discussed in Step 3.

In viewpoint models, these dependency relations typically exist between concepts and properties referring to *different* entities, but describing the *same* property. For example, in the left part of Fig. 7, concepts *bike* and *money* both have a value, while they describe different entities that depend on each other. In partial models these dependency relations typically exist between *different* properties referring to the *same* entity. For example, in the right part of Fig. 7 the concept *mountain bike* contains a property *value* that depends on the property value of *delivery time*, both properties refer to the same real-life mountain bike entity.

The results of this step are as follows:

- Identification of intra-model dependency relations between concepts and properties,
- Dependency relations are categorized in (1) symmetric and asymmetric, and (2) in property and existence dependencies.

5.6.2 Step 6: intra-model consistency constraints

Goal: Identify intra-model consistency constraints for each model. More specifically, we use the intra-model dependency relations identified in Step 5 to formulate constraints for each relation.

Analogously to defining inter-model consistency constraints in Step 4, we define intra-model consistency constraints in Step 6. Note that these constraints still assume the models are built properly, i.e., that they are well-formed. Intra-model constraints depict constraints on concepts, properties, and their relations within the model, not on the language used to describe the model. For example, there is no constraint stating that concepts can only be connected through a specific arc since this is a language constraint. However, there exist intra-model constraints stating that a specific concept shall be present if another one occurs. For example, if a payment is done, a bike needs to be shipped because this is a model-specific constraint (i.e., a specific constraint for our example).

Dependency relations identified between the different concepts in Step 5 are used to define the constraints. The same structure for building the constraints as in Step 4 is applied. As difference, constraints are now defined *within* a model and not *between* models.

In addition to consistency constraints, which are defined based on dependencies *between* concepts, the *existence* of the concepts itself, with or without a specific property value, is defined here. For example, if a concept represents money transfer from customer to company, this concept should also *occur* in the implementation, regardless of dependencies with other concepts. The general form of such a constraint is as follows:

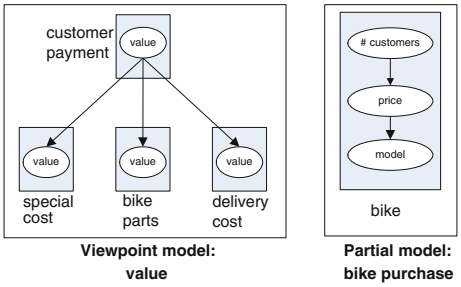

Fig. 11 Generalization over intra-model consistency constraints

– Let x be a concept with property y in a model. The corresponding consistency constraint states: x exists with property y.

For example, if concept *money transfer* with property value *100 euro* exists in a model, the constraint states this concept with property value needs to exist in the implementation.

Often it is possible to generalize over different consistency constraints as it is done for inter-model consistency constraints in Step 4. By generalizing over a set of constraints, the number of constraints to be checked reduces significantly, increasing applicability of our method. Next, we discuss for both viewpoint models and partial models which consistency constraints are generalized, and how this is accomplished.

Viewpoint models. In Step 5, intra-model dependencies are identified. In this step, each of them is used to formulate a consistency constraint. As a result there are many similar consistency constraints:

Example The left part of Fig. 11 depicts a viewpoint model that describes monetary values of different entities. For example, *bike parts* have a certain monetary value, and *delivery* costs are paid by the company to the delivery service. Properties (i.e., monetary values) are dependent on each other. For example, the total price (i.e., property value) for the customer depends on the price of the bike parts.

Generalization is done for constraints that depict the *same type of dependency* and only over constraints being *interconnected* through concepts and properties. This is the same approach as taken for generalization in partial models for inter-model consistency constraints.

Example There are three constraints for the value viewpoint model (cf. left part of Fig. 11): 1) *costs for the customer depend on special costs*, 2) *costs for the customer depend on bike part costs*, and 3) *costs for the customer depend on delivery costs*. These constraints are generalized into a general one that states: *Total costs for the customer depend on the sum of special costs, bike part costs, and delivery costs*.

In general, we state that generalization in viewpoint models is possible if the dependency relations are *interconnected*, and if they connect the same type of *property*. For asymmetric relations it should be possible to form a *tree* out of the relations (as done in the example), and for symmetric ones it should be possible to create an *undirected graph* from the relations.

Partial models. Dependencies in partial models identified in Step 5 typically exist between properties within one concept that refer to the same entity. We illustrate generalization in partial models:

Example The right part of Fig. 11 depicts a partial model for the concept *bike*. It has three properties: *expected number of customers*, *price*, and *model type* (e.g., mountain bike). The number of customers depends on the price of the bike and the price, in turn, depends on the model type. Each of these dependencies results in a consistency constraint. For generalization we develop one consistency constraint that captures both dependency relations, i.e., relations between number of customers and price, and between price and model type. The constraints state: 1) *For a mountain bike (i.e., a specific model) x euro are paid*, and 2) *If a customer pays x euro for a bike, y customers will buy one.* The general one now states: *For a mountain bike the price is x euro, and y customers buy the bike*.

Generalization within partial models is possible over those constraints that describe relations between different properties of one concept (i.e., one entity) if these properties are interconnected. For asymmetric relations, like in our example, it should be possible to build a tree with the properties and their relations. For symmetric relations, in turn, it should be possible to build an undirected graph.

Joint constraints. Each model is built with a specific *purpose*. It prescribes (1) which entities should occur, (2) which properties they should have, and (3) in which way they should manifest in the cooperation. Typically these are *model generic* constraints where a model depicts, for example, (1) which transfers occur between actors in one business transaction, (2) what the value of certain exchanges between actors is, or (3) the order in which entities should occur in a business transaction. Model-specific consistency constraints are formulated taking the model purpose into account. It is not possible to formulate a general constraint for this purpose since this is model-dependent. Generalization is often not necessary since the constraints are typically model-specific and therefore general.

The results of Step 6 are as follows:

– Identification of intra-model consistency constraints, and
– Illustration of generalization of consistency constraints.

5.7 Phase IV: combined analysis

5.7.1 Step 7: dependency analysis

Goal: Formalization of model parts that are checked for consistency, and formalization of the defined intra-model and inter-model consistency constraints. This formalization enables easy implementation for automatic consistency checking.

Checking consistency is typically done by *testing* models with some model checker, or by finding a *translation*. Since we not only check consistency between models, but also between models and running system, we choose to translate them. Typically, models are translated into a semantically well-defined formalism, which enables formal consistency checking. Either complete models are translated or only their overlapping parts. In the inter-model and intra-model analysis phase (Phase II and Phase III) we identify crucial parts for consistency checking, i.e., we identify overlapping parts. Therefore, we choose to *partially translate* models into a *language-independent*, formal notation, i.e., we only translate those parts crucial for consistency checking. Formalization of model parts and their constraints enable easy implementation. This facilitates automatic constraint checking at runtime in the Management Phase (cf. Steps 8 and 9). To manage models of inter-organizational cooperations, we monitor several parts of the models:

- concepts with their properties,
- relations between properties, and
- relations between concepts.

We monitor these items by checking consistency constraints as defined in Steps 4 and 6. These constraints depict dependencies between concepts and between properties. We check whether the constraints hold by comparing running system and models. We illustrate what is present in the formal models with the following examples:

Example The value viewpoint model in the left part of Fig. 7 consists of two concepts with a *value* property that have a symmetric dependency relation. The concepts and their constraint (based on the dependency relation) are included in the formal model. Further, all concepts in the message viewpoint model have some dependency relation, and are, therefore, related to one or more constraints. Therefore, all concepts and their constraints in the message viewpoint are part of the formal model. This also holds for constraints describing symmetric relations between *bike* and *delivery*, and between *money* and *payment*, respectively.

Example Concepts *mountain bike* and *bike parts* in Fig. 7 are included in the formal model since their properties have dependency relations. Further, inter-model consistency constraints describing the asymmetric property dependency relations between the two prices and the two delivery dates, respectively, are included. Also intra-model consistency constraints (i.e., property dependency between price and delivery date of the bike and the bike parts, respectively) are included.

Many formalizations are possible. The key is to *group* representations of concepts that belong to the same business activity. For example, using sets, one instance is represented as a set, while using graphs one instance is represented as a graph:

Example In the left part of Fig. 7 concepts *bike* and *money* are grouped since one purchase of a bike entails both bike and money transfer. Furthermore, also concepts *order, confirm, send, deliver*, and *pay* are grouped since these messages together make up one purchase of a bike. Regarding the right part of Fig. 7, value properties and delivery time properties are grouped.

The results of Step 7 are as follows:

- Formalization of model parts that are used in the consistency constraints, and
- Formalization of consistency constraints.

5.8 Phase V: management phase

In the management phase, we check whether or not the running system performs as prescribed in the models. In other words, we check consistency between event logs and formal models, i.e., the dependency models. To enable such comparison, we analyze event logs and create an *event log model* that enables easy comparison in Step 8. The result of this comparison is reflected in the *management models* where inconsistencies are depicted. These models enable *causal analysis* to identify why inconsistencies occur (Step 9). Using this analysis, different solutions for handling inconsistencies are identified. However, as a consequence of applying these solutions, new inconsistencies might be introduced. It is important to identify these consequences because we aim at *minimizing* the number of model changes when regaining consistency between models and running system.

5.8.1 Step 8: log analysis

Goal: Abstract necessary information from the event logs to monitor the models and their constraints.

In inter-organizational models conceptual structures are present that are not visible in event logs. To check them for evidence of consistency between models and system behavior, we need to add this structural information. Therefore, we

suggest to reconstruct relations between event log entries. For example, for each entry in the event logs we identify to which instance it belongs: If a payment is done and afterward stored as entry in the event log, it is necessary to identify for which service or product this payment is done.

Identifying these relations is a widely known problem for which different solutions exist: Many approaches aim at deriving this structural information through mining techniques [2], while other approaches add structural information to event logs when they are created [29,42]. We do not describe data mining or event log structuring in detail. Rather, we discuss which information is necessary for our approach.

We use *identifiers* to reconstruct model structures captured by event logs. We assume that the structure as described in the models is reflected in both implementation and event logs. Furthermore, we assume identifiers are added to an event log entry. For example, a *contract number* can be used to identify to which transaction an entry belongs, *log on* and *log off* messages can be used to identify separate transactions, and *timestamps* (e.g., "Thu, 17 July 2009 09:23:12") can be used to identify in which order certain events occur. For each cooperation it needs to be decided which identifiers can and need to be used.

Necessary information abstracted from event logs is structured using *sets*. Event log entries belonging to one transaction are grouped into a set. Each entry is represented as a *tuple*. Typically, each tuple contains a timestamp, issuer, recipient, unique name, and some property information. The resulting model contains all information necessary for checking constraints in the formal models. In general, we construct the event log model as follows:

1. Identify which property values should be present in the event log model using the dependency models.
2. Represent each entry in an event log as tuple containing a timestamp, issuer, recipient, unique name, and all property values.
3. Group tuples belonging to one transaction into sets using identifiers like log-on and log-off messages, and contract numbers.

After creating the event log model, we compare runtime results with the models describing the system. The result is represented in *management models*. When using sets to represent the formal models, consistency constraints are checked by *matching* tuples and sets between event log model and formal model. When using graphs, tuples and sets of the event log are matched with vertices and edges of the graphs. Furthermore, additional constraints are matched by checking occurrences of tuples, properties of tuples, and occurrences of sets in the event log model. For example, if a consistency constraint states that the presence of a concept (i.e., tuple) indicates the presence of another one (i.e., tuple), this relation is checked in the different sets of the event log model.

Example In our example, the company offers two business transactions: purchasing mountain bikes and purchasing city bikes. At runtime, we collect information from event logs. For example, we expect evidence of selling mountain bikes, represented as sets of events representing the sale. In addition, we expect evidence of selling city bikes in a similar manner. Furthermore, to sell bikes, the company needs to purchase its parts before. This constraint is checked during runtime. We expect evidence to support the constraint that bike parts are purchased before bikes are sold, represented as events with different timestamps.

Violation of consistency constraints is different for existence dependencies and for property dependencies. When a constraint describing an *existence dependency* is violated, this constitutes a violation where the concept does not exist. For example, a bike needs to be paid by the customer after he receives it. If there is no evidence of payment in the event log, this is a violation of the constraint. If a constraint describing *property dependency* is violated, this violation can come in gradations, depending on the scale of the value of the concept. For example, when the selling price of a bike is twice the amount paid for the bike parts according to the model, and the runtime result in the event log deviates from this ratio.

The results of Step 8 are as follows:

– A description of information that needs to be abstracted from the event logs for monitoring the models and their constraints.

5.8.2 Step 9: causal analysis

Goal: Identification of causes for constraint violations, and identification of consequences of restoring consistency between models and running system.

The monitoring results containing constraint violations need to be presented in an intuitive way. We suggest to represent monitoring results by showing deviations in *color codings* (cf. Fig. 14, Sect. 6.2 as example). For example, *red* indicates violation of an existence or property dependency constraint, *green* indicates compliance, while *orange* indicates a deviation of not more than 10%. Coloring can be done in original models or in dependency models. In this way, both arrows (indicating dependencies) and concepts (containing property values) are colored. The results are management models that show relations between concepts, whether these relations are violated during runtime, and whether their property values are accomplished.

If management models are created, it becomes clear which parts of the running system comply with the models and

which parts do not. If there is a violation the analyst can either evolve the *models* or the *running system*. In this paper we focus on changing models so that they reflect the running system.

Violations are often related. A constraint violation in one part of the model often results in violations in other parts of the model. These causal relations are important to identify for efficient model management since solving a constraint violation of the source might solve numerous other violations at the same time. The dependency models created in Step 7 show all dependencies between concepts. Here, these dependencies are used to identify causes for violations. For this causal analysis *only asymmetric dependencies* are used because in those relations it is clear which concept influenced the other. In symmetric dependencies you might travel to end leafs of the problem instead of identifying the source of the violations:

Example If there is an existence constraint violation of concept *money* in Value viewpoint model A (cf. left part Fig. 7) (i.e., there is no evidence that money has been paid), the symmetric dependency relations make it difficult to determine its cause. For example, the symmetric relation between concepts *bike* and *money* does not indicate whether money is not paid because the bike is not delivered, or the other way around. In other words, determining causes in symmetric relations is difficult.

After deciding which model parts to change, it is important to identify what *consequences* these changes have. For example, when making a product more expensive to resolve a lack of income, the number of expected customers might decrease, leading to even less income. Therefore, besides identifying *causes* for violation, it is also important to decide *which parts* to change.

Every element in the constraints can cause an inconsistency and is, therefore, able to regain consistency. Often, there are different ways to make a change in one of the models to regain consistency. For example, if during runtime *bike parts* turn out to be more expensive than agreed upon in the models, one solution is to negotiate a lower price, while another possibility is to change provider and purchase bike parts elsewhere. Although both solutions solve the constraint violation, the first one is less intrusive for the model than the second one, since the second solution results in deletion of an offered service. To distinguish intrusive from less intrusive changes, we divide possible model changes into different categories. Each constraint violation is now solved by applying a subset of changes suggested by these categories. Each change has its own consequences for the models.

- *Non-observable changes* in a model have no impact on the formal model, i.e., the change does not influence the dependency models. As a consequence, these changes are outside the parts that influence consistency. For example, in the right part of Fig. 7 the partial models describe the mountain bike and its parts. If the payment method changes, this does not affect these models since payments are not captured.
- *Observable structural changes* in a model are changes where concepts are removed or added to the model while preserving its well-formedness. These changes influence *existence dependencies* since these dependencies rely on the existence of certain concepts. Also *property dependencies* are influenced since non-existence of a concept implies that its property value does not exist. For example, if we remove concept *bike* in the left part of Fig. 7, this will be an observable structural change. It affects the existence dependencies with concept *money* of the same viewpoint model, and with concept *delivery* of the message viewpoint model B.
- *Observable non-structural changes* are changes where the property value of a concept is affected, or where the way two concepts are related is affected. For example, when changing the order of two concepts, their relation is affected. These changes do not affect constraints on existence dependencies since they do not change the existence of concepts. However, it does affect constraints on property dependencies since it changes property values. For example, if the *value* property of concept *bike parts* in the right part of Fig. 7 becomes larger, i.e., the bike parts become more expensive, then this will be an observable non-structural change. It affects also the *value* property of concept *mountain bike* since the price of the mountain bike depends on its parts.

Explicating for each constraint which changes can be used to regain consistency (if necessary) is a step toward more efficient and precise model management. By relating changes to constraints, it is possible to predict the impact of a change, not only on the violated constraint, but also on other constraints that are related to this particular change.

With this last step we conclude the analysis of models and event logs. Using our method, it is possible to *monitor* consistency constraints between models and between models and running system. Further, it is possible to identify *causes* for violations based on a causal analysis. With the analysis, violated constraints are selected for inconsistency resolution. Now, we describe the last part where we use identified possible changes and dependency relations to predict *consequences* of changes made in the models.

Each suggested change to regain consistency might affect more than the violated constraint. By considering dependency models, we identify which constraints are affected when changing a particular concept. An observable structural change affects both the existence and property dependencies, while an observable non-structural change only affects

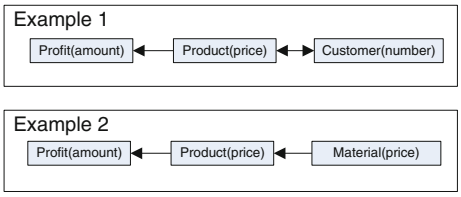

Fig. 12 Consequence analysis for changing models

related property dependencies. If the affected dependency is symmetric, the change also affects the related concept. If the affected dependency is asymmetric, the change only affects the related concept in case it depends on the original one:

Example Examples 1 and 2 (cf. Fig. 12) depict a property dependency model where the constraint between *product price* and *amount of profit* is violated. In other words, event logs show that the property relation between product and profit does not hold. The model can be changed to solve this inconsistency: Profit, product price, or ratio between profit and price can be adjusted.

Assume the developer considers changing the product price. Now, consequences for other dependencies are analyzed. In Example 1, *product price* and *number of customers* depend on each other. Therefore, if the *product price* changes, the *number of customers* is affected, just as their dependency relation. In Example 2, *product price* depends on its *material price*. Now, changing *product price* does affect their relation, but not the *material price*. The developer concludes that changing product price in the second example has limited consequences, while changing product price in the first example leads to more adjustments.

Especially in the context of large models and many dependencies it is useful to enable analysis where changes are related to types of dependencies. Analysis is done for the consequences of such changes in the rest of the models, but also for constraints it has with other models. Our method uses constraints, dependencies, and types of changes to show which parts of a model are affected by the specific change. As a result, the developer is better able to estimate necessary effort to adapt models, and he is better able to make a choice between different change possibilities. In general, the most suitable change is the one with the least impact on other constraints.

The results of Step 9 are as follows:

- We provide a method for causal analysis of the dependency models created in Step 7. This analysis identifies possible causes for constraint violations.
- Furthermore, we provide a method to predict the impact model changes have on consistency constraints. This supports analysts in identifying the least intrusive model adaptation to regain consistency.

6 Evaluation

In Ref. [5], we validate our method by conducting two *case studies*. To show suitability for a variety of models, the two scenarios are sufficiently different. The first one captures *business and coordination models*, and the second scenario concerns *Service Level Agreements* of composite services.

6.1 Scenario 1: business and coordination models

Bodenstaff [5] shows applicability of our method regarding models of inter-organizational cooperations. We consider two fundamental perspectives which are of high relevance for modeling such cooperations: *business* and *process perspective* [25,1]. At business level [25], expectations of exchanged values, like e.g. expectations on the number of transferred products and their prices, between partners are modeled as a value model. In particular, the exchange of goods, money, and immaterial goods (like e.g. customer satisfaction) between different actors are modeled. Further, estimates are made on the expected size of market segments, number of customers, sales per customer, as well as prices for products and services. This information is used to determine whether an inter-organizational cooperation is beneficial for all actors involved. At process level ([1]), coordination of inter-organizational processes is modeled as a coordination model. In particular, messages exchanged between different partners, and the order of exchanged messages, thus the supported business transactions are represented. The model is the basis for describing observable behavior of information systems operated at various partners.

In case of perspectives the aim of the MaDe4IC approach is to keep models consistent, i.e., the models describe the same information system. Event logs, as considered in the MaDe4IC method, represent run-time observable behavior of involved information systems, thus event logs are an abstraction of the underlying information systems. In addition, business and process models must be related to event logs to identify inconsistencies between models and actual implemented information systems. Understanding dependencies between business and process model and event logs can be facilitated to resolve model inconsistencies.

Business and process perspectives used in this scenario describe necessary transfers between partners although focusing on different aspects (finances versus coordination). Since both models have a different level of abstraction, use different modeling notations, and have a different purpose, determining consistency is a challenge.

Following our MaDe4IC method intra- and inter-model consistency constraints are identified (Steps 4 and 6) and dependencies with the event log are derived (Step 8). These constraints and dependencies are now facilitated (Step 9) to determine the effect of a change on one model on the

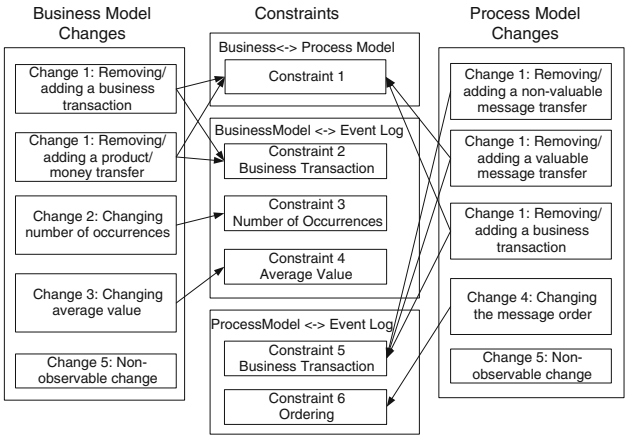

Fig. 13 Relating consistency constrains and model changes

constraints and, therefore, its potential implication on other models.

Figure 13 illustrates the business and process model changes as well as the identified constraints between models and event log as a basis for the causal analysis in Step 9 (for a detailed description see [7]). A main bilateral constraint between all models is that the business transaction of one model has to be consistent with the business transaction in the other model (see Constraints 1,2, and 5). Further, explicit model capabilities must be consistent with the event log, i.e., number of occurrences and average value of an exchanged object for business models (Constraints 3 and 4), as well as the order of exchanged messages in business models (Constraint 6) must be consistent with the event log. The changes on business and process models are represented as basic change operations on models altering specific properties of the models. The class of unobservable changes (Change 5 in business and process model) refers to changes in the model, which are not effecting any constraints. An unobservable change is for example the change of an internal order approval step within a business model effecting only a single partner. The arrows between changes and constraints indicate that changes in a model effect consistency constraints. Usually a change is performed to resolve an inconsistency. However, if a change has several outgoing arrows, then a change may resolve inconsistency on one constraint while introducing an inconsistency on another constraint. These dependencies between changes and consistency constraints can be facilitated in a change management tool for maintaining consistent models.

6.2 Scenario 2: service compositions

Also in [5], we present a second scenario, which is inherently different from the first one. We apply our method to monitoring Service Level Agreements (SLAs) for service compositions. A typical example of such an SLA is:

In 90% of all cases, invocation of service X will have a response time within y milliseconds (ms).

We use SLA models to describe quality of service of a composition. More specifically, we focus on monitoring response times and costs of services. The challenge is to relate different SLAs, taking the composition into account. Further, we develop an approach to manage SLAs at runtime.

For a business operating in a networked environment it is vital to accurately manage services it provides to its customers. This is particularly challenging if a company offers *composite* services where interactions with services offered by other providers influence its performance. Consider a composite service which returns combined information from several search engines. In this case, the quality of service (e.g., response time) that can be offered depends on the quality of service delivered by the search engines. Together with constraints of the customer, these calculations form the basis for the SLA between customer and service provider [28,45].

Since SLAs are typically bilateral agreements, current monitoring approaches (e.g., [28,45,51]) focus on identifying violations in bilateral communication. However, to properly manage its composite service a company has to reason about *causes* of SLA violations. For example, if the offered response time for a composite service provided by a company depends on response times of other services this company uses, it is vital to identify and monitor these *dependencies*. Exactly these dependencies are ignored in bilateral monitoring approaches. With our MaDe4IC method for managing dependency relations in inter-organizational models we developed the *MoDe4SLA approach* (MOnitoring DEpendency relations for SLAs) ([6,8]). With this approach, we can analyze during development phase different types of *dependencies* between services, and the *impact* these services have on each other. Further, it allows combining bilateral monitoring results with analyzed dependencies and impact services have on the composition.

By applying our MaDe4IC method to the problem of managing service compositions, we are now able to manage the complex dependencies between the different SLAs of composite services. We manage these dependencies by creating a *feedback tree* (i.e., a dependency model) that shows deviations between agreed upon SLAs (i.e., constraints) and real-life behavior of services (cf. Fig. 14), i.e., this tree shows SLA *violations*. Furthermore, *causes* for violations are detected through dependencies between services (cf. Step 9).

The feedback tree depicts deviations from agreed upon SLA values. Colors on edges and vertices are used to visualize these deviations. Currently, red, green, yellow, darkgreen, and colorless are used, but these colors can be extended or changed in any preferred way. Intuitively, red and yellow represent negative deviations, while green and darkgreen represent positive deviations. The goal of the feedback tree

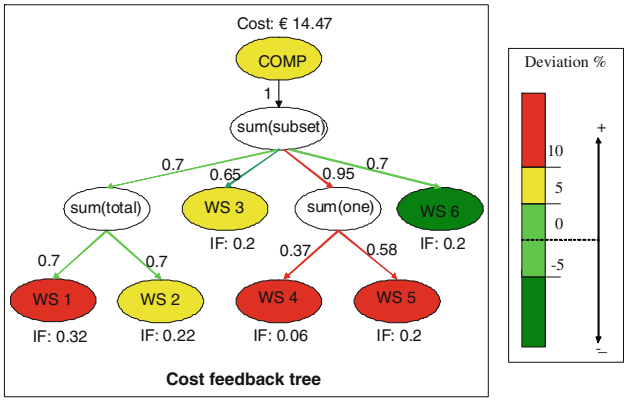

Fig. 14 Illustrative example: Cost feedback tree

is to support management in identifying causes for badly performing compositions.

The result of applying our method to service compositions is illustrated by a feedback tree (cf. Fig. 14) as generated by MoDe4SLA. The tree shows a service composition that depends on six Web services concerning its costs. Using the event log abstraction, it is calculated that the average cost for invoking the given composition is 14.47€. This is between 5 and 10% under performance, and, therefore, colored yellow. Further, we see that only Web service WS 6 costs on average less than expected, i.e., the service is colored darkgreen. The other services all cost on average more per invocation from that agreed upon (i.e., colored red or yellow). We see that the branch with WS 3 is invoked less often than expected, i.e., is colored darkgreen, as opposed to the branch of WS 4 and WS 5 which is invoked more than expected, i.e., is colored red. WS 4 has a low impact on the overall cost (IF 0.06), while other Web services have an impact between 0.2 and 0.32. MoDe4SLA enables users to

- identify how well each service is performing,
- detect whether services are invoked as often as expected,
- determine the impact each service has on the composition regarding its SLAs.

In the given case, we derive that the exceeded composition costs are caused by its underlying services (i.e., we do a causal analysis, Step 9). These services, in turn, exceed their agreed upon invocation costs. Although WS 4 violates its costs SLO, its impact is low. Further, the expensive branch (with WS 4 and WS 5) concerning costs is more often invoked than expected. The cheaper branch (i.e., the branch with WS 3) concerning costs, in turn, is invoked less often, causing the overall composition to exceed its agreed upon costs. Our MaDe4IC method proves especially useful for identifying the complex relations between different services. Furthermore, it provides the means to develop an approach for causal analysis to identify causes for SLA violations.

7 Summary

In this paper, we introduce a stepwise method toward efficient model management of inter-organizational cooperations. We develop our MaDe4IC method by investigating general properties of inter-organizational models and by conducting an extensive literature research. The approach itself relies on the identification of different types of dependencies between the different concepts within and between models. Furthermore, we suggest translating relevant model parts into formal models for easy consistency checking. The same approach as used for these formalizations is applied to abstracting useful information from event logs into an event log model such that runtime behavior of the system can be compared with the models describing it. Furthermore, an additional causal analysis, using identified dependencies, allows identifying causes for inconsistencies between models and running system. As a last option the method provides an approach to check the consequences for consistency constraints when changes are made to a model. In other words, we analyze what the consequences are for other consistency constraints when trying to resolve an inconsistency. As a whole, our method presents a stepwise, structured approach into managing these complex constellations in an effective and efficient way. The method is evaluated by conducting two case studies from different research domains where the method is applied.

References

1. Aalst WMPvd (2000) Loosely coupled interorganizational workflows: modeling and analyzing workflows crossing organizational boundaries. Inf Manag 37(2):67–75
2. Aalst WMPvd, Dongen Bv, Herbst J, Maruster L, Schimm G, Weijters A (2003) Workflow mining: a survey of issues and approaches. Data Knowl Eng 47(2):237–267
3. Andrews J, Zhang Y (2003) General test result checking with log file analysis. IEEE Trans Softw Eng 634–648
4. Astesiano E, Reggio G (2003) An attempt at analysing the consistency problems in the UML from a classical algebraic viewpoint. In: Proceedings of WADT'03, Springer, pp 56–81
5. Bodenstaff L (2010) Managing dependency relations in inter-organizational models. PhD thesis, University of Twente, to be published soon: Ask guest editor for a copy
6. Bodenstaff L, Wombacher A, Reichert M, Jaeger MC (2008a) Monitoring dependencies for SLAs: The MoDe4SLA approach. In: Proceedings of SCC'08, pp 21–29
7. Bodenstaff L, Wombacher A, Reichert M, Wieringa R (2008b) An approach for maintaining models of an e-commerce collaboration. In: Proceedings of EEE'08
8. Bodenstaff L, Wombacher A, Reichert M, Jaeger MC (2009) Analyzing impact factors on composite services. In: Proceedings of SCC'09

9. Bowman H, Boiten E, Derrick J, Steen M (1996) Viewpoint consistency in ODP, a general interpretation. In: Proceedings of FMOODS'96, pp 189–204
10. Bowman H, Steen M, Boiten E, Derrick J (2002) A formal framework for viewpoint consistency. Form Methods Syst Des 21(2):111–166
11. Dadam P, Reichert M (2009) The ADEPT project: a decade of research and development for robust and flexible process support. Comput Sci Res Dev 23(2):81–97
12. Derrick J, Boiten E, Bowman H, Steen M (1996) Supporting ODP – translating LOTOS to Z. In: Proceedings of FMOODS'96, pp 399–406
13. Dustdar S (2004) Caramba-A process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams. Distribut Parallel Databases 15:45–66
14. Easterbrook S, Chechik M (2001) A framework for multi-valued reasoning over inconsistent viewpoints. In: Proceedings of ICSE'01, pp 411–420
15. Easterbrook S, Finkelstein A, Kramer J, Nuseibeh B (1994) Co-ordinating distributed viewpoints: the anatomy of a consistency check. Technical Report 94/7
16. Egyed A, Medvidovic N (2000) A formal approach to heterogeneous software modeling. In: Proceedings of FASE'00, Springer, pp 178–192
17. Engels G, Küster J, Heckel R, Groenewegen L (2001) A methodology for specifying and analyzing consistency of object-oriented behavioral models. SIGSOFT 26(5):186–195
18. Engels G, Heckel R, Küster JM, Groenewegen L (2002) Consistency-preserving model evolution through transformations. In: Proceedings of UML'02, Springer, pp 212–226
19. Feather M (1998) Rapid application of lightweight formal methods for consistency analyses. IEEE Trans Softw Eng
20. Finkelstein A, Kramer J, Nuseibeh B, Finkelstein L, Goedicke M (1992) Viewpoints: a framework for integrating multiple perspectives in system development. Int J Softw Eng Knowl Eng 2(1):31–58
21. Finkelstein A, Gabbay DM, Hunter A, Kramer J, Nuseibeh B (1993) Inconsistency handling in multi-perspective specifications. In: Proceedings ESEC'93, pp 84–99
22. Fradet P, Métayer DL, Périn M (1999) Consistency checking for multiple view software architectures. SIGSOFT 24(6):410–428
23. Frank U (1999) Conceptual modelling as the core of the information systems discipline—perspectives and epistemological challenges. In: Proceedings of AMCIS'99, pp 13–15
24. Giunchiglia F, Shvaiko P, Yatskevich M (2004) S-Match: an algorithm and an implementation of semantic matching. In: Proceedings of ESWS'04, Springer, pp 61–75
25. Gordijn J, Akkermans H, van Vliet H (2000) Business modelling is not process modelling. In: Proceedings of eCOMO'00, Springer, pp 40–51
26. Hunter A, Nuseibeh B (1998) Managing inconsistent specifications: reasoning, analysis, and action. ACM Trans Softw Eng Methodol 7(4):335–367
27. Jensen K (1997) Coloured petri nets. Basic concepts, analysis methods and practical use, three volumes. Springer, Berlin
28. Keller A, Ludwig H (2003) The WSLA framework: Specifying and monitoring Service Level Agreements for Web services. J Netw Syst Manag 11(1):57–81
29. King S, Chen P (2005) Backtracking intrusions. ACM Trans Comput Syst 23(1):76
30. Kleene S (2002) Mathematical logic. Dover, New York
31. Loucopoulos P, Zicari R (1992) Conceptual modeling, databases, and CASE: an integrated view of information systems development. Wiley, New York
32. Mens T, Van Der Straeten R, Simmonds J (2005) A framework for managing consistency of evolving UML models. In: Software Evolution with UML and XML, pp 1–31
33. Mitchell T (1982) Generalization as search. Artif Intell 18(2):203–226
34. Nentwich C, Capra L, Emmerich W, Finkelstein A (2002) xlinkit: a consistency checking and smart link generation service. ACM Trans Internet Technol 2(2):151–185
35. Nentwich C, Emmerich W, Finkelstein A (2003) Consistency management with repair actions. In: Proceedings of ICSE'03, pp 455–464
36. Noy N, Musen M (2001) Anchor-PROMPT: using non-local context for semantic matching. In: Proceedings of ONTOL'01, pp 63–70
37. Nuseibeh B, Easterbrook S, Russo A (2000) Leveraging inconsistency in software development. Computer 33(4):24–29
38. OMG (2009) OMG UML specification. http://www.omg.org/technology/documents/formal/uml.htm
39. Osterwalder A, Pigneur Y (2002) An e-business model ontology for modeling e-business. In: Proceedings of Bled electronic commerce conference'02, pp 17–19
40. Paolucci M, Kawamura T, Payne T, Sycara K (2002) Semantic matching of Web services capabilities. In: Proceedings of ISWC'02, Springer, pp 333–347
41. Pokraev S (2009) Model-driven semantic integration of service-oriented applications. PhD thesis, University of Twente
42. Porras P, Neumann P (1997) EMERALD: event monitoring enabling responses to anomalous live disturbances. In: Proceedings of national information systems security conference'97, pp 353–365
43. Rahm E, Bernstein PA (2001) A survey of approaches to automatic schema matching. VLDB J 10(4):334–350, http://link.springer.de/link/service/journals/00778/bibs/1010004/10100334.htm
44. Rozinat A, Aalst WMPvd (2008) Conformance checking of processes based on monitoring real behavior. Inform Syst 33(1):64–95
45. Sahai A, Machiraju V, Sayal M, van Moorsel APA, Casati F (2002) Automated SLA monitoring for Web services. In: Proceedings of DSOM'02, pp 28–41
46. Sefika M, Sane A, Campbell R (1996) Monitoring compliance of a software system with its high-level design models. In: Proceedings of ICSE'96, vol 18, pp 387–396
47. Smith R (2007) Aristotle's logic. Stanford encyclopedia of philosophy
48. Spanoudakis G, Zisman A (2001) Inconsistency management in software engineering: survey and open research issues. In: Handbook of software engineering and knowledge engineering, vol 1, pp 24–29
49. Teorey T, Yang D, Fry J (1986) A logical design methodology for relational databases using the extended Entity-Relationship model. ACM Comput Surv 18(2):197–222
50. Timmers P (1998) Business models for electronic markets. Electron Markets 8:3–8
51. Tosic V, Pagurek B, Patel K, Esfandiari B, Ma W (2005) Management applications of the Web Service Offerings Language (WSOL). Inf Syst 30(7):564–586
52. Uchitel S, Chechik M (2004) Merging partial behavioural models. SIGSOFT 29(6):43–52
53. Varró D, Pataricza A (2003) VPM: a visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. J Softw Syst Model 2(3):187–210
54. VDA (2005) VDA Recommendation 4965 T1: engineering change management (ECM)—part 1: engineering change request (ECR) version 1.1. Association of German Automobile Manufacturers (VDA)

55. Wand Y, Weber R (2002) Research commentary: information systems and conceptual modeling—a research agenda. Inf Syst Res 13(4):363–376
56. White S (2008) Business Process Modelling Notation (BPMN). http://www.bpmn.org/Documents/BPMN November 2, 2009
57. Yao Y (2004) Granular computing. Comput Sci 31:1–5