Integration – Reflections on a Pivotal Concept for Designing and Evaluating Information Systems

Ulrich Frank

University of Duisburg-Essen, 45141 Essen, Germany
ulrich.frank@uni-duisburg-essen.de
http://www.wi-inf.uni-duisburg-essen.de/FGFrank/

Abstract. Integration is one of the pivotal concepts with respect to analyzing, designing and evaluating information systems. Integrating software components as well as integrating information systems with the surrounding action system is a core activity of most software development projects. Despite the outstanding importance of integration in Information Systems, only little work has been done on developing a conception of integration. This paper presents a conception of integration that can be applied to information systems as well as to their integration with the action systems they are supposed to support. It allows for differentiating degrees of integration. It also accounts for additional levels of integration, such as meta level or instance level integration.

Keywords: Coupling, integration, IT business alignment, redundancy, reusability, semantics.

1 Introduction

The benefits of integration are probably part of every curriculum in Information Systems and Computer Science. Integration helps to avoid redundancy, hence, contributes to integrity. It is also regarded as an effective means to accelerate business processes and to support decision making. Catering for integration is a pivotal objective during the development of information systems. On the one hand, it suggests designing a common conceptual foundation, e.g. an object model, for the applications or components to be integrated. On the other hand, it recommends the use of system components that are dedicated to integration, such as Database Management Systems, so called Middleware, Workflow Management Systems etc. In addition to the build-time stage, integration is a major issue during system maintenance. A plethora of projects is aimed at healing the problems caused by isolated, heterogeneous software systems. Since these systems are often still vital for a company's performance, IT managers tend to avoid their removal. However, ex post integration faces severe challenges. Numerous consulting companies and tool vendors have responded to this demand by offering methods and systems, referred to as Enterprise Application Integration, Data Warehouse Systems, Middleware etc.

While suprisingly little publications are available that target integration explicitly – two of the rare examples are (cf. [3], [1]) – numerous research activities are aimed at fostering both, ex ante and ex post integration of information systems. Research on conceptual modeling targets the development of modeling languages and methods that allow for developing a common conceptual foundation for integrating the components of an information system. Research on ontologies is intended to support the development of integrated information systems, too ("ontology-driven development", (cf. [4]). Ontologies are also suggested for re-engineering purposes in order to integrate heterogeneous applications (cf. [2]; [4]). Federated databases are a further research field that is focused on ex post integration.

In contrast to the vast amount of research that is focused on the integration of information systems, it is surprising that there is hardly any publication available that is aimed at developing a conception of integration. This may be contributed to the fact that integration is regarded as a core term, which does not need further explanation. However, such an implicit notion of integrity is not sufficient for designing or evaluating information systems. Therefore, I will attempt to develop a conception of integration that should serve this purpose. It is rather a reconstruction of the actual use of the term than a proposal for an entirely new concept. In the first part of the study, a preliminary concept serves to analyze various aspects of integration and related benefits. Subsequently, a refined concept will be presented, which allows differentiating degrees and levels of integration.

2 Integration: Aspects and Benefits

The term integration represents both, a process and its result, i. e. a feature of a system or a set of systems. The focus in this paper is on integration as a system feature. According to the colloquial meaning of the term, integration results from constructing a whole from previously isolated parts. However, such a general conception is not sufficient to account for the peculiarities of information systems. Also, the image of hardware parts that can be composed only, if they have fitting physical shapes, is not appropriate. Information systems are not physical, but linguistic artifacts. Integrating two linguistic artifacts requires that they are able to communicate – either directly or through some kind of mediator. In other words: If two components of a system are not able to communicate, they are not integrated. Note, that I use the term 'component' as an abstraction of software artifacts such as applications, modules, or components in a more specific sense. Communicating means exchanging data, e.g. by sending messages or by using shared memory. However, catering for data exchange only is not sufficient: Communication can work only, if both components share a common interpretation of the data. Otherwise, communication would be a threat to efficiency and to system integrity. Against this background, we can outline a first draft of what we mean by integration: Two components of an information system are integrated, if they can communicate. If they have the ability to communicate but do not make use of it, they are *potentially integrated.*

2.1 Dimensions of Integration

To cope with the complexity of large systems during systems analysis and design, it is common to make use of abstractions that focus on particular aspects only. Static or data abstractions, captured typically through data models, reduce a system to a description of its data structures. Functional abstractions focus on the functions, a system is supposed to perform. Data flow diagrams would be an example of a functional abstraction. Finally, dynamic abstractions represent the control flow within the processes of a system. They could be realized through state charts or – on a higher level of abstraction – through business process models. Object-oriented abstractions can be regarded as a composition of static and functional abstractions. These three essential abstractions of information systems can be used for illustrating different dimensions of integration.

Static Integration
Our preliminary concept of integration can be applied directly to this dimension. Two components of an information system are statically integrated, if they can exchange data – either directly or through a mediator. For this purpose, statically integrated components need to share common concepts that specify the semantics of the exchanged data, e.g. common data types, classes or other data structures. Apparently, static integration allows for sharing data and – as a consequence – for avoiding data redundancy. However, static integration does not imply data sharing. It only requires common concepts that define the semantics of the exchanged data. If, for instance, two components exchange data, they may do this only once in a while and still manage their own – redundant – data in the meantime. In this case, however, there would be a better chance to protect data integrity, e.g. through some kind of synchronisation protocol, because both components would share the same meaning of duplicate data. If two applications have access to common data but do not make use of it, they would be regarded as potentially integrated. The benefits of static integration are obvious. Firstly, there is no need for hazardous and expensive interpretations of data that origin from other components. Secondly, if data are shared by both components, there is no need for updating different copies. Apparently, both aspects contribute to the integrity and efficiency of an information system. Note that the proposed concept of integration requires at least one integrated component to write data that is being used by the other components. Hence, applications that have read-access only to some common resource files provided by the operating system are integrated with the operating system, but not between themselves. Also, a so called data warehouse does not contribute to the integration of legacy applications, since the database that is aggregated from operational level data, is usually not being used by operational level systems. The operational level systems do not share the schema of the data warehouse. Nevertheless, a data warehouse creates the impression of integrated data for those tools that use it. We could call it virtual integration.

Functional Integration
Communication that is restricted to somehow exchanging shared data does not allow for directly addressing another component of the system. This is different with functional integration. Two components of an information system are functionally integrated if they communicate through calling functions. This requires them sharing the meaning of those functions. At the same time, it requires them to be statically integrated. Only then can they use the data that is required or delivered by the functions that were called. Functional integration allows the participating components to share the functions they have in common. Therefore it contributes to avoiding redundant functions, which in turn facilitates system maintenance. In order to reduce the number of communication channels, a central mediator can be deployed. It can either dispatch function calls to a prospective receiver or to a central function library. Note, however, that functional integration is not accomplished, if two components use a common library without exchanging data. If the functions allow for write-access, it will at least be a case of potential integration. While functional integration implies static integration, data integration will often depend on functional integration as well: If, e.g. two components communicate through a common file system or a common DBMS, they both need to share the required access functions. A specific asset of functional integration is the ability to establish data exchange between components that do not share the same data representation – but only the same interfaces. This allows for leaving existing data structures unchanged, which is a specific benefit for achieving ex-post integration. There is, however, a severe downside to this approach: integration of this kind does not cater for avoiding data redundancy – a problem that is often ignored by proponents of service-oriented architectures. Hence, functional integration can contribute to "covering the mess" instead of removing it.

Dynamic Integration
Functional integration allows two components to cooperate in the sense that they mutually use/provide services. However, it is not sufficient for the integrated components to jointly execute a certain process. This is subject of dynamic integration. It requires a common collaboration or process schema, which includes common event types. Furthermore, it implies functional integration. A process schema defines a process e.g. through event-action-rules: If a certain event is generated by one of the integrated components, it will trigger the appropriate component to execute a certain function. Such a decentralized execution plan requires all the integrated components to have access to the common process schema – and to identify the components that are in charge of executing a particular function. Taking into account that a component may participate in more than one process schema, this approach will often imply a remarkable effort for the specification and implementation of the components. It corresponds to the coordination of autonomous software artifacts ("agents") according to a common plan. To avoid this effort, integration of the components can be relaxed by deploying a central coordinator. Relevant events generated within a

component would then have to correspond to common event types defined in a central schema. In addition to that, the components would have to be functionally integrated.

Organizational Integration
Orthogonal to the dimensions discussed above, there is a further dimension of integration that is of particular relevance for Information Systems: The efficient use of information systems requires them to be integrated with the organizational action systems they are supposed to support. This dimension has gained remarkable attention both in business practice and academia, often referred to as "IT-Business-Alignment". To define a conception of organizational integration, we build on the previous definitions: An information system and the actions system it is supposed to support are integrated, if they are able to communicate. This requires common or better: corresponding concepts, since the specification of concepts in information systems is different from natural language terms. If, for instance, a database schema includes the concept of a "PreliminaryInvoice", integration suggests that this is an established term of the action system as well. Otherwise users cannot use this component of the system properly. Organizational integration can be differentiated into static, functional and dynamic. Static refers to communication that is based on common static concepts, such as the above example. Functional integration refers to the concepts that specify functions which correspond to functions or tasks within the action system. Finally, dynamic integration is aimed at coordinating functions provided by the information system and human actions according to a process schema. It requires events generated within an information system that correspond to concepts, users are familiar with. On the other hand, events within the action system that are relevant for the information system should also correspond to a common concepts. There are different approaches to accomplish organizational integration. Firstly, the concepts of an information system can be adapted to the concepts used in the action system. In this case, requirements analysis is aimed at identifying the terminology of the intended application domain in order to map it to corresponding concepts for designing the information system. Note that 'terminology' refers mainly to concepts, not to their designation. Secondly, the prospective users can adapt to the information system's concepts. This does not only require learning the concepts but also mapping them to existing terminology and – eventually – adapt the existing terminology and corresponding patterns of action. Thirdly, there is the option of mutual adaptation, which will often be a convincing choice: With respect to user acceptance, it is not a good idea to ignore existing terminology and established patterns of action. At the same time, both existing terminology and established patterns of action are often insufficient with respect to actual and future business requirements. Hence, organizational integration does not only require analyzing existing domains and their prevalent language. Instead, it suggests (re-) designing new action systems and reconstructing existing terminologies that are in line with corresponding information systems – thus contributing to the evolution of new language games.

3 A Refined Conception of Integration

While our first conception of integration includes the distinction of various dimensions, it does not allow for differentiating different qualities or intensities of integration. Therefore, I will suggest a refinement of the concept that accounts for different degrees and levels of integration.

3.1 Degrees of Integration

In general, integration requires common concepts that provide a foundation for communication. However, with respect to facilitating communication not any common concept is of equal value. For communication to be efficient and consistent, common concepts should allow for specifying the exchanged data in a way that minimizes information loss. In other words: The amount of semantics, a common concept includes, should not be lower than that of any of the corresponding concepts in the components to be integrated. Note that this use of the term "semantics" corresponds to information content: The more possible interpretations are excluded by the specification of a concept, the higher its semantics. This allows for refining the concept of integration to express different degrees of integration: The higher the amount of semantics incorporated in common concepts used by a set of integrated components, the higher the *degree of integration* of these components. If, for instance, two components exchange bytes, the extent of possible interpretations is apparently larger than for the exchange of data, which are defined as floating point numbers. Referring to a concept such as "Customer" or "Invoice" would promote an even higher degree of (static) integration. This concept can be applied to all dimensions of integration. The semantics of a function is more difficult to describe than that of data. On the one hand, it depends on the semantics of its interface: A function that is passed a number is less specific – hence allows for more interpretations – than a function that is passed a more complex structure. Also, additional constraints, e.g. expressed through pre- or postconditions, increase a function's semantics. The degree of dynamic integration depends on the semantics of event action rules that define coordination. The semantics of event-action-rules depends on the semantics of the involved event types and the semantics of the functions that represent the actions. An event type such as "file was modified" allows for more interpretations than "product database was updated" or even "price of product x was changed". Fig. 1 illustrates various degrees of integration in all three dimensions.

The refined concept of integration can also be applied to organizational integration. If, for instance, customer information is stored in text files using a text processor, the level of organizational integration is low: the common semantics of "Customer" is reduced to a string. If an accounting system features a concept of "Account" that directly corresponds to the use of the term in the intended application domain, the degree of integration would be higher – in other words: the chance for communication failures would be less. If an ERP system includes an elaborate concept of "OrderProcessing" that is not known in its application

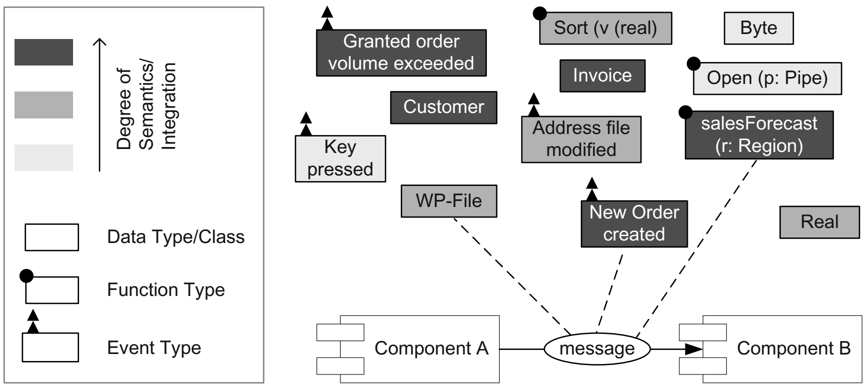

Fig. 1. Degrees of Integration

domain, the level of integration would – at first – be low, too. Only, if system introduction and training resulted in successful adoption of the concepts suggested by the ERP system, integration could be accomplished. The higher the degree of organizational integration, the better is the chance to avoid friction between human action systems and the supporting information system, in other words: the better is the alignment of business and IT. Apparently, there is a difference between the dimensions of system integration and organizational integration: Due to the interpretative flexibility and the learning capabilities of human users, a perfect fit is not necessary. Due to the interpersonal diversity of meaning, it would not be detectable either. While the refined concept of integration allows for gradually differentiating degrees of integration with respect to certain concepts, it does not allow for specifying the degree of an entire information system. If two information systems have only one highly specialized concept in common, would they feature a higher degree of integration than two other system that share a number of concepts with less semantics? Against this background, it might be tempting to define a formal concept of integration that would allow for calculating corresponding metrics, e.g. the average or total degree of integration of an entire information system. However, while such an approach might be regarded as satisfactory by some, it is likely to produce distortion. The notion of semantics we use is not a pure formal one. For instance: With respect to formal semantics, there is no difference between typing an attribute "name" as String and typing an attribute "revenue" as String. However, accounting for the meaning, we contribute to the term "revenue", it would clearly indicate a poor degree of integration, if two components used String as a common concept.

3.2 Levels of Integration

So far, our focus was on concepts that are required to facilitate integration. Integration, however, can also be related to the instance level – and to meta levels as well.

Instance Level Integration
Communication requires common concepts. The concepts can be refered to using a common namespace, e.g. the types of an IDL. This allows for assigning semantics to instance level data that is exchanged between the components of an information system. However, for communication on the instance level to be efficient, refering to corresponding concepts alone is not sufficient. A unified identification, hence a common namespace, would not only help reducing communication load – it would be sufficient to exchange identifiers only – it would also allow for differentiating between instances. For this reason, it makes sense to expand the previous notion of integration by accounting explicitly for the instance level: Two components are integrated on the instance level, if they share a common namespace to identify instances. Integrating instances implies conceptual integration, i. e. common concepts to specify the semantics of the instances. Instance level integration through common namespaces is an important feature of so called middleware or DBMS. These technologies allow for identifying data objects accross the boundaries of specific namespaces maintained by separate components. Apparently, this corresponds to static integration. Integrating function instances is a different issue. At first, it is not as obvious what a function instance is: Is it a particular implementation, a particular implementation within a particular context or a particular execution? There may be many instantiations of a particular implementation. Therefore, regarding a particular implementation as an instance, makes sense only for functions that provide stateless services. A service offered by an object is a prototypical example of a particular implementation within a particular context. The identification of such a function requires the identification of the corresponding object. Hence, instance level integration of functions is an issue only in those cases, where the static context is relevant; in other words: instance level integration of functions implies static integration of those elements that constitute the relevant context. Core concepts required for dynamic integration are events, functions and corresponding execution rules. Particular events are generated e.g. through the instantiation of data or objects, their deletion or state changes. If, in a cooperation scenario, e.g. a workflow, the coordination of two components demands for taking into account particular events, there is need for dynamic integration on the instance level. If, for instance, in a workflow management scenario, an incoming order is represented through an object, changes that apply to this particular object – e.g. representing the approval of the order – are relevant particular events. In order to accomplish this kind of integration – either in a decentralized cooperation scenario or through some kind of mediator – it is required to define a namespace for event instances that allow for idendifying events throughout the entire workflow (and maybe within an even wider context). The benefits of instance integration are obvious: It is the precondition for avoiding redundancy and for different components working jointly on common concepts. Achieving instance level integration can require a remarkable effort. Not only that it requires to maintain a common namespace, which covers distributed components. It also requires appropriate instruments for instance lifecycle management. In the case

of static integration, this includes assigning a common name (identifier), when an object is instantiated and ensuring referential integrity when it is removed. This is similar for function instances: new instances need to be registered and assigned an identifier. In case, a function instance is removed, referential integrity is an issue, too. In the case of dynamic integration, instance life cycle management is even more demanding. There is, e.g. need for a protocol that defines, when an event is regarded as consumed, hence, needs to be removed from the common event space. To summarize, instance level integration faces two main problems: consistently identifying and managing instances accross the set of components that are to be integrated. The safest approach to consistent identification and naming is the use of unambiguous identifiers for the real world entities to be represented in a system. If every relevant object carried its identification – like the identification number in a passport or a number used to identify a specific instance of a product type – there would be no longer the need to care for consistently assigning identifiers to instances in information systems. If these objects would also carry the definition of their type, assigning real world objects to representations within information systems would be even more efficient and safe. In the long run, this may even result in overcoming the distinction between a real world object and its representation within an information system: Every object could carry a representation of itself, together with a unique identifier. Given the standards, bandwiths and affordability required to realize such a vision, a tremendous impact on the economics of logistics processes can be expected.

Meta-level Integration
If – in human communication – we use terms other participants do not know, we can still explain these terms using other, common terms. The effort such an explanation takes depends on the available common language. If, for instance, the concept "Interface" as it it used in the UML needs to be described, it will be much easier, if one can refer to the technical terminology of software engineering than to colloquial language only. A similar approach to communication is possible in information systems, too. For example, the concept of an "Interface" is specified through a meta model. Components that are able to interpret the concepts of the corresponding meta model, could infer the semantics of the concept, even if it not part of a schema they have access to. However, the UML meta model is hardly sufficient to specify the semantics of particular classes, e.g. of a class "Product". The meta level concepts would only specify that it is a class with certain features. A further example would be an XML document representing an invoice that is transmitted together with its DTD. While the receiving component – if it includes an adequate interpretation of XML documents – would be able to identify that the document is a valid XML document, it could not infer that the DTD describes the concept "Product". If one used a more specific language on the meta level, there would be the chance to describe concepts more specifically, leaving less room for interpretations. For instance: A formal "Business Language" could include a meta concept such as "Product", which would capture the essence of all product types within a certain range.

As a consequence, every component that has a reference to this meta concept could infer that all instances of this meta concept represent product types. For a comprehensive example and its application to prevalent architectures see ([5]). Note that in addition to common meta level concepts there would be need for unified designators – which might be multilingual.

4 Challenges

The obvious advantages of integration are constrasted by reservation. On the one hand, warning voices can be heard from those who are in charge of managing IT. They doubt the economic benefits of integration. On the other hand, there are software engineers who advocate a more relaxed concept of integration in order to better cope with the demand for flexibility.

4.1 Complexity and Risk

Designing and implementing information systems that feature a high degree of integration is certainly more demanding than building isolated systems with a more modest degree of integration: It is not sufficient to strive for abstractions that fit a particular application. Instead, abstractions should result in concepts that fit a whole range of applications. In order not to jeopardize the integrity of the system during its lifecycle, abstractions should be resistant against changing requirements. With the number of components to be integrated the likelihood that abstractions fail in this respect is growing. Ex post integration is even more challenging, since it requires touching legacy systems. Demanding a high degree of integration in this case requires modifying existing, often insufficiently documented code. From the perspective of those who are responsible for integration projects, this is a rather threatening scenario, since numerous sources of hidden risks can be expected. While there is not doubt that building integrated systems is associated with remarkable complexity and risk, that does not necessarily discredit the striving for a high degree of integration. One should take into account that reducing complexity usually requires increasing it first. Building, e.g., a DBMS is certainly a complex endeavour. If it was successful, however, it contributes to a remarkable decrease of complexity – and effort – for those who build applications. Therefore, from an economic point of view, the additional effort for realizing a higher degree of integration should be related to the corresponding benefits. This will certainly require a thorough evaluation, e.g. by analyzing the effect of integration on current and future business processes. There is, however, no doubt that especially cross-organizational integration is facing a paradox situation: On the one hand, it offers exciting economic perspectives. On the other hand, the incentives for pioneering actors to invest into cross-organizational integration are often not sufficient.

4.2 Integration Versus Loose Coupling

In recent years, loosely coupled systems seem to be the name of the game in software development. However, coupling is not a well defined term. On the one hand, it refers somehow to the principle of information hiding: Two components are loosely coupled, if their mutual dependance is restricted to a well-defined interface. Hence, the internal representations of the components are not affected. On the other hand, coupling refers to the specifity – or semantics – of these interfaces. If a component's interface is specified on a low level of semantics, it is more likely to find further components that can be coupled to it. Hence, loose coupling suggests to use interfaces only to integrate systems and to avoid a high degree of integration. At first sight, such a recommendation may seem useful. However, it has clear shortcomings. Firstly, integrating two components through concepts with little semantics is a clear threat to system integrity. Secondly, reducing integration to interfaces, i. e. abstracting from the internal representations of the involved components, leaves an extremely relevant issue open: data redundancy. Note that this is different with information hiding in object-oriented software development: Each object is thought to represent a unique real world entity. Therefore, the data represented in objects should not be redundant.

Apparently, there is a tradeoff between the benefits of integration and the threats of dependance. Deciding for the right balance in a particular case recommends taking into account two aspects. Firstly, one should analyse whether the essence of the concepts used to accomplish a higher degree of integration is the same for all participating components. If this is the case, the changing requirements related to some components would not affect the common concepts. In other words: One should put emphasis on the quality of the abstractions used to accomplish integration. Secondly, one should not neglect the concepts used in other systems that one might want to integrate in future times. While both aspects create a remarkable challenge, it is not a good idea to sacrifice the benefits of integration for the supposed sake of loose coupling without further consideration.

5 Concluding Remarks

When it comes to designing and evaluating information systems, integration is a pivotal term. This recommends a thorough conceptualisation and assessment. The ideas presented in this paper are preliminary only. They leave many questions open. Also, a concept that is based on "semantics" adopts some of the unsolved mystery that is still encompassing this term. Currently, integration is an ambivalent concept. Its clear benefits are contrasted by threats. This is similar to reusability, which marks the other side of the same coin: The more semantics a reusable artifact incorporates, the larger its benefit in a certain reuse case, the higher, however, the likelihood that is does not fit a particular context. Unfortunately, it is not possible to overcome this conflict through invidual action. There is, however, a promising approach to developing a solution to this unsatisfactory situation: The creation of *reference models* (cf. [7]; [6]) that serve as blueprints for an entire range of information systems. Unfortunately, the development and establishment of reference models exceeds the resources available to single academic institutions by far. Also, it is required to involve prospective users and vendors in time. Traditional approaches to organizing research on information systems are not appropriate for that. Instead, there is need for large communities that bundle their forces not only to develop and evaluate reference models, but also to promote them. A recent initiative, started by the SIG MobIS within the German Informatics Society (GI), is aimed at applying the idea of Open Source Software to the development of open reference models ([8]), (www.openmodels.org). While it is a demanding process to establish such initiatives, it seems the only chance to exploit the potentials offered by integration and reusability.

References

1. Anderson, N.H.: Foundation of information integration theory. Academic Press, New York (1981)
2. Bouras, A., Gouvas, P., Kourtesis, D., Mentzas, G.: Semantic Integration Of Business Applications Across Collaborative Value Networks. Springer, Boston (2007)
3. Wiederhold, G. (ed.): Intelligent Integration of Information. Kluwer Academic Publishers, Boston (1996)
4. Guarino, N.: Formal Ontology and Information Systems. In: Guarino, N. (ed.) Formal Ontology in Information Systems. Proceedings of FOIS 1998, Trento, Italy, June 6-8, 1998, pp. 3–15. IOS Press, Amsterdam (1998)
5. Frank, U.: Modeling Products for Versatile E-Commerce Platforms – Essential Requirements and Generic Design Alternatives. In: Arisawa, H., Kambayashi, Y., Kumar, V., Mayr, H.C., Hunt, I. (eds.) ER Workshops 2001. LNCS, vol. 2465, pp. 444–456. Springer, Heidelberg (2002)
6. Frank, U.: Evaluation of Reference Models. In: Fettke, P., Loos, P. (eds.) Reference Modeling for Business Systems Analysis, pp. 118–140. Idea Group, Hershey (2006)
7. Fettke, P., Loos, P.: Classification of reference models - a methodology and its application. Information Systems and e-Business Management 1(1), 35–53 (2003)
8. Frank, U., Strecker, S.: Open Reference Models – Community-driven Collaboration to Promote Development and Dissemination of Reference Models. Enterprise Modelling and Information Systems Architectures 2(2), 32–41 (2007)