Technology-Independent Modeling of Service Interaction

Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
Email: gerald@cs.auckland.ac.nz

Abstract

Systems based on a service-oriented architecture (SOA) can be implemented with many different technologies, and in particular, they can be implemented with a heterogeneous set of technologies. An enterprise service bus (ESB) is a typical option for bridging the technology boundaries. It is desirable to have technology-independent models of the core services in the IT system. We present here computation-independent models (CIMs) and platform-independent models (PIMs) for service oriented architectures. Our models have the following advantages: Some of the CIMs are closely related to Petri net approaches; the PIMs are expressed in the same formalism as the CIMs; a canonical PIM is easily derived from a CIM; the semantics of the PIMs matches the operation of a typical enterprise service bus architecture. Finally, both CIM and PIM are defined as core semantic data models and can therefore be created with most semantic data modeling tools.

1 Introduction

In this paper we discuss technology-independent models for message-based communication of information systems. One key idea of a model-driven approach is to use platform-independent models that can be translated into several platform-dependent models, thus enabling reuse. For service-oriented architectures, many frequently discussed languages are not platform-independent; BPEL, for instance, is tailored towards web-services. In the same vein, it is important to realize that a (mis-)understanding of service-orientation as mere migration to web-services is an implementation technology and not an architecture. Today's enterprise computing projects are covering areas as diverse as healthcare [7] and e-commerce [15]. In such projects, a plethora of different message-based technologies is used; compare for example e-commerce with classical EDI [10] and AS2 [12], a novel e-commerce standard that is similar to web-services [4, 14]. In order to understand such diverse systems is helpful to assume a technology-independent viewpoint. The reference model for ODP defines a viewpoint as "a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system." [1]. We will propose our own viewpoints, which are mostly related to the enterprise viewpoint of ODP; our viewpoints will then motivate the different technology-independent models that we introduce. We introduce computation-independent model (CIMs) and platform-independent models (PIMs). More important than the individual labeling of these models as either CIMs or PIMs is however that they are in a clear semantic relationship to each other.

In Section 2 we discuss one of our key modeling principles, namely the heavy use of immutable datatypes, and derive CIMs that are not yet tied to a datamodel. In Section 3 we introduce the key aspects of our definition of core data models. In Section 4 we explain another key modeling principle, namely the use of datamodels to represent business integrity constraints that have a process-like character. We introduce CIMs based on this approach. In Section 5 we introduce CIMs that express synchronization. In Section 6 we reflect on the importance of message-based communication for enterprise computing. The message-based viewpoint presented in Section 7 captures the high-level architecture of state-of-the-art systems, particularly enterprise service bus architectures. Here we introduce our platform-independent models. In Section 8 we revisit the advantages of using core data models. In Section 9 we summarize our findings.

2 The memo-based viewpoint

Our first viewpoint is the *memo-based viewpoint*. In this system view, all IT-related business activities proceed by creating documents and locking them at a certain time. After that they are immutable and remain in the sys-

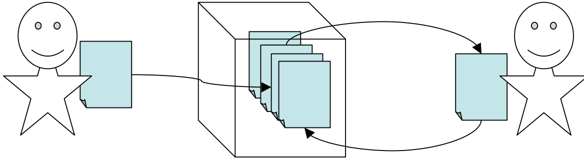

Figure 1. Memos are kept in a repository.

tem state. We call these immutable, archived documents *memos*. Hence the memo-based viewpoint models archiving on an analysis level as the repository of memos, as illustrated in Figure 1. User-generated as well as relevant system-generated memos should be kept. The first type of computation-independent model are therefore data models that contain only memos, we call them *memo models* (MMs).

The memo-based viewpoint is an enabling viewpoint for later process-oriented and message-based models. Memos are a precursor to messages, but in memos the aspect of transporting information is not yet emphasized, while persistence is emphasized more.

One remarkable semantic aspect is that the immutability constraint for the memos is inseparably connected with a timestamp concept. Every immutable object has a unique timestamp where it becomes immutable. We want to call this process *locking* in order to indicate that there could possibly be a previous edit phase. The locking of an immutable data object is an atomic event, and this means that the immutability constraint asks naturally for a transactional concept. This again gives rise to the importance of submit/response style systems.

The semantic essence of the memo model is the collection of memos, that allows only inserts of new memos, and the association of a timestamp with each memo. The concept of memo models is however not bound to a particular data model for the description of the individual memos. Therefore we understand memo models as being compatible with for example textual memos as well as with memos that consist of structured data. The space of versions of a wiki would constitute a memo model. The memo-based viewpoint is significant because via the immutability constraint and the ensuing differentiation of distinct timestamped memos it structures the information space.

The way we introduced the concept of timestamps for every memo is an example of the way we understand analysis in the software engineering process: analysis can add rich attributes like timestamps even where they might not be supported by every modeled system (i.e. the system might keep no, or unreliable timestamps) in order to foster system understanding.

The memo model allows strictly no change of the memo content. Extensions of this concept that in turn allow amendments of submitted data, akin to a concept of minor edits, are thinkable but can often be naturally defined on top of the memo model concept.

3 Core data models

The memo-based viewpoint does not require a certain data model, however for the upcoming process-oriented viewpoints we will use a relational model to describe the interconnection of memos. Therefore we employ a data model of the relational family. In our view this family especially includes conceptual models such as ER and some parts of UML. We, however prefer a semantically more minimalist approach and therefore only use core data models. A core data model for our purposes is a data model that has entity types and binary relation types, and the relation types have set-relational semantics. Generalization as a separate notion is not required. In form-oriented analysis [6] we call our modeling language the *parsimonious data modeling language*. Its models are called PD models for short, they are core data models to begin with. The PD models can be seen as simplifications of UML and ER models. A PD model in our approach is described by a mathematical object, its so-called model graph. Because we conceive PD model graphs as mathematical objects, the visual representation is not prescribed. We prefer a visual notation which is similar to graph notation. A PD model consists of *entity types*, *relation types*, and *roles*. An entity type has entity instances. A relation type is a predicate which says whether a number of entity instances are thought of as being connected. Each role *role(a, t)* is connected with one relation type a and one entity type t. The *model graph* is defined to be a tuple (E, P, R, e, p), where E is the set of entity types, P is the set of relation types, and R is the set of roles. $e : R \mapsto E$, $p : R \mapsto P$ are the functions giving for each role its entity type and its relation type. The *static semantics* of a PD model are the set of all the possible *states* of this model. An entity type represents the collection of all *entity instances*. Entity instances are opaque identities. Each entity type is assumed to be a *repository*, i.e., a countably infinite set of entity instances. We call an entity instance that has never been used before a *fresh entity instance*. In PD models the primitive data types are just entity types, and there is no difference between attributes of a type and relation types. In each state a relation type is represented by a finite relation between the connected entity types.

The *dynamic semantics* of a PD model describe the possible updates on the state of the PD model. Updates are the transition from one state to another. This definition gives rise to a typed automaton model of the system. The system changes its state through updates, while the PD model remains the same. Here we distinguish two important notions of updates: primitive updates and transactions. The

primitive updates are simply the insertion or the deletion of a single link between two entity instances. The operation *insert of link l on relation type a, insert(a, l)* means that a becomes $a \cup \{l\}$. The *delete of link l on relation type a, delete(a,l)* means that a becomes $a \setminus \{l\}$. The insert operations can make use of a new() operator that delivers fresh identifiers. The *transactions* are complex updates which are still executed atomically. They will be clarified through an automaton model later.

4 Using core data models for modeling business logic

The key semantic approach in this paper is that we will model process-like aspects of the business logic with core data models. Memos in a business process are often based on earlier memos. As an example let us consider two types of memos in a banking application, a memo type representing the opening of a new account and a memo type representing the granting of a personal loan. Both memos are related to an important business process in a bank, the granting of personal loans. A business constraint is, for instance, the constraint that a loan must only be granted after an account has been opened. This is a process-like constraint in the business logic.

In order to model such process-like constraints we use relation types with the following additional constraints. We consider *arrow heads* as annotations on roles of relations. The entity type at the annotated role is the target and the other entity type is the source. A *time* arrow head is defined on relation types between timestamped entity types. The defining condition of a time arrow head is that the timestamp for each target instance t is not earlier than the timestamp of any source instance that is linked to t via the annotated relation type. We call these source instances the predecessors of t and t the successor of them. In other words, time arrow heads always point in positive time direction. A relation type with a time arrow head is a *time relation type*. We will consider here only time relation types between memos. A set of time relation types has the property that in every state the directed links of these relation types form a directed acyclic graph. We call such constraint *partial oder* constrains. Im UML, aggregations can be used to some extent, but the time constraints are not implied by aggregations.

In the rest of the paper we will model process aspects with time relation types between memo types. The personal loan example is modeled by a time relation types between the two memo types so that the loan memo is the successor. A 1..1 multiplicity on that time relation type at the role of the account creation memo type makes sure that every loan grant has to refer to one earlier account creation. This example illustrates that we can model process aspects with time

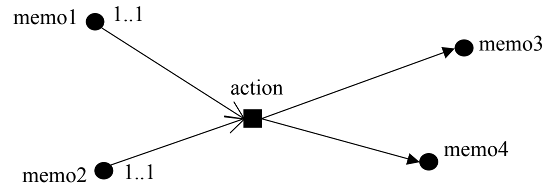

Figure 2. A Memo Flow Model

relation types in core data models alone, without the need for a separate process language. The time relation type and the multiplicity alone ensures that the intended process must be followed. If $t_1, t_2, \ldots, t_n$ are the generation times of the different memos in the process, then our approach enforces $t_1 \leq t_2 \leq \ldots \leq t_n$. The immutability of memos is applied here as well.

This use of time relation types gives rise to a computation-independent model for process-like aspects, and we call this the *memo order model* (MOM). This model only contains memos and time relation types between these memos. In the following we will see that we can create more specialised models for processes based on this. These models will be specializations of MOMs. The previous definition of the more general MMs is helpful, because there are specialization of MMs other than MOMs. A strongly typed reimagination of email for example would only support single precursor memos, either of the Re: style or of the Fwd: style. Indeed the DTIMs introduced later, which are platform-independent models, use such an approach. Wikis on the other hand, where memos are page versions, not pages, also support only one predecessor memo, with the additional constraint that the predecessor relation establishes a total order. The concept of a changeable page is a special case of a topic bundle [6], this is an identifier named in a memo as a topic. For wikis we usually have the additional constraint that each page version names only one such identifier as topic and all page versions that name this topic are part of a single total order of memos. In Figure 1 the right hand actor performs a wiki style edit. It has the benefit that it looks like a change to the old page, hence appealing to intuition, but it is in fact the creation of a new immutable page, without destruction of the old version.

5 The action viewpoint

An often discussed phenomenon is synchronization between subprocesses. Not every new memo must lead immediately to a subsequent action. Assume, we need a quote and an approval in order to confirm a travel booking. Only if both messages have arrived, a confirmation can be cre-

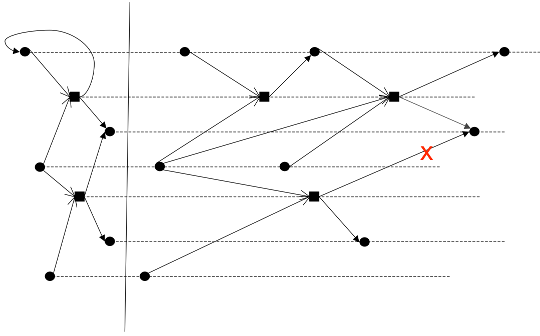

Figure 3. Data model and example state of a memo flow model.

ated. We therefore want to define a viewpoint that expresses this abstraction, the *action viewpoint*. Actions that are performed get their own identity, and they can result in several memos being produced. An action type is an entity type and it has relation types to all memos that are required and all memos that are produced. The action viewpoint results in a bipartite action model resembling a Petri net as shown in Figure 2. The two partitions are actions and memos. The actions are depicted as square nodes, the memos are depicted as round nodes. The actions themselves do not hold additional data, but they help in defining what we call a superparameter [6], that is a data type that acts as a parameter list. This is in contrast to many languages where a parameter list might be a type of the type system, but is not a first class citizen data type. All the memos linked to an action are input to that action. In this model, it is in general not necessary that the action is performed immediately as soon as all necessary predecessor memos are present. Such a constraint would however be frequent for individual action types. The action viewpoint gives rise to a computation-independent model (CIM). We call a data model that shows memos and actions a memo flow model (MFM). A memo flow model is a directed bipartite graph; this is a well-formedness condition on memo-flow models.

Multiplicities that refer back from an action to predecessor memos describe which predecessors must be there at the start of the action. If for example the multiplicities are all 1..1 multiplicities as in Figure 2, then they represent a synchronization, and the action behaves similar to a synchronization bar in a Petri net.

5.1 Semantics of memo flow models

In Figure 3 a memo flow model is shown on the left and an example state over this model is shown, by indicating with (here horizontal) swimlanes, to which type each entity instance is belonging. This type of diagram was introduced in form-oriented analysis to discuss semantic properties of models (The diagram works best if the relation types are unique between two entity types). A first semantic property that we require from relation types in a MFM is that they are time relation types. This means, that the state over the data model is a directed acyclic graph. A memo flow model, as a directed bipartite graph, has two types of edges: action-memo edges and memo-action edges. The second semantic property only affects the action-memo transitions; these are the arrows going from actions to memos. The requirement is that every memo is only created by one action. This is equivalent to UML composition (in UML composition the diamond would be on the opposite end of the relation type from the arrowhead given in the MFM). This property is illustrated in Figure 3. Note the edge that is crossed out. The targeted memo is also produced via an edge from a different action. One edge must be deleted. In contrast, one action can be targeted by an arbitary number of memo-action transitions, if no other constraints prevent that.

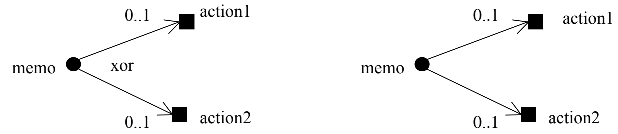

Figure 4. Nondeterminism in a Memo Flow Model

5.2 Nondeterminism and the action viewpoint

The action viewpoint is reminiscent of advanced Petri nets since it is bipartite. In contrast to Petri nets however, there is no implicit consumption of messages by their successor messages. This becomes obvious if we consider nondeterminism as in the following examples. In the two small MFMs in Figure 4, in the left picture each message is giving rise to either action1 or action2 but not both. In this sense the message is consumed, but only because of the xor constraint (note here that the xor constraint and the multiplicities are stated following the PD model convention, not the UML convention). On the right-hand side, however, the message may give rise to two actions because of the multiplicities. The reason we prefer MFMs over sophisticated Petri net variants is the tight integration of MFMs with core data models.

6 Importance of message interchange

Message-based data interchange is used in many mature technologies. EDI is an implementation technique for

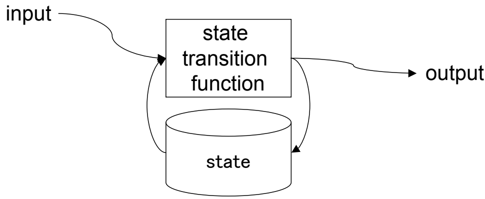

Figure 5. Unit systems are modeled as automata

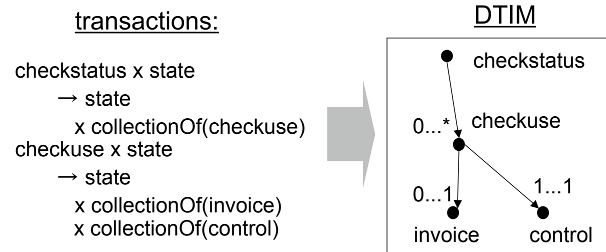

Figure 6. DTIMs fit to the typed automaton model.

business message interchange [8]. It allows communication with high Quality of Service. The interchange is traditionally on dedicated networks, called value-added networks. The more recent technology of web services [2] is an implementation technique for message communication. Web services use XML as a semi-structured format [3] for messages. Notations for distributed systems based on Web services include Web service orchestration languages, for example BPEL. Web services use a type system given through the XML Schema concept. Deployed Web services are described by the Web Service Description Language WSDL. This language can be used for the specification of configuration information, i.e., for the data transmission options chosen. This approach is technology-dependent: BPEL primarily describes Web service communications. A higher degree of abstraction is needed for modeling at the design or analysis stage.

7 The message-based system viewpoint

The *message-based* viewpoint allows us to capture the architecture of state-of-the-art message-based systems. In this viewpoint all memos are considered to be messages. Each model in this viewpoint is called a *data type interchange model* (DTIM). DTIMs are high-level design models for message-based communication. They fit well for example to enterprise service bus architectures [11], but also to similar systems used in electronic data interchange [5].

The core architectural feature of such systems is that the business logic is composed of components that are triggered by messages. This common high-level design might be implemented with various concrete component technologies, such as message-driven enterprise java beans (EJBs) or XML style sheets.

7.1 Unit systems as automata

In form-oriented analysis, distributed systems are conceived as a net of single systems, called *unit systems*. Such a unit system is a *computational automaton* with a state as illustrated in Figure 5. This automaton takes in messages and produces other messages. In an untyped view, such an automaton is specified by a single *state transition function*:

$$\texttt{stateTransitionFunction:}\ \ \texttt{message} \times \texttt{state} \\ \rightarrow \texttt{state} \times \texttt{collectionOf(message)}$$

In the statically-typed view, each transaction has an associated message type. The state transition function is conditional and invokes for each message type a different *transaction*, which is the state transition for this message type. In Figure 6 on the left-hand side, two transactions for the messages *checkstatus* and *checkuse* are shown.

We can therefore represent the transactions by their message types. A single message type acts as the superparameter of the transaction. Each such transaction may represent one deployed transformation component from Figure 7, such as workerA or workerB. Message-based components working on persistent message queues are an old type of component, known from classical transaction monitors. A recent name for a directly analogous technology of message-based middleware today is enterprise service bus (ESB). The automaton model fits very good to the general ESB architecture; the transactions are equivalent to worker threads in such a system as illustrated in Figure 7. They are crucially important for a workable enterprise platform, and it was a major drawback for recent object-oriented enterprise platforms that they did not have message-driven components from the start; their later addition was eagerly awaited by practitioners. Such message-based components are an exact replica on the implementation level of our platform-independent concept of transactions. Indeed such message-based components follow the superparameter concept, in that the message is their sole parameter and it is of course a structured data object. As an additional side remark one might add that these concepts bear a resemblance to the coordination language Linda [9], but in practice the central requirements for users are nonfunctional requirements, chiefly persistence of the messages. There is also

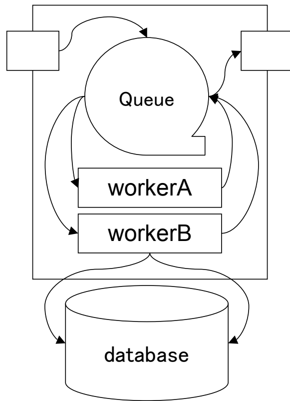

Figure 7. Transactions are analogous to components in an ESB architecture.

a direct correspondence with active database technologies: the transaction for message m is in principle a trigger on insert on table m, and it follows the event-condition-action pattern. The transactional execution style used here is called *detached* in active database terminology.

7.2 Data type interchange models

The message-based viewpoint motivates the platform-independent Data Type Interchange Models (DTIMs) that we are going to introduce now. They are first of all a natural translation of the above mathematical definition into a data model. The DTIM in its elementary form contains as entity types (depicted as nodes) just message types that represent the transactions as explained before.

If the transaction for message type A may send messages of type B, then the DTIM contains a time relation type from A to B. The connection between the mathematical notation for a single transaction and the node in the DTIM is shown in Figure 6. Again, since the edges are just relation types, DTIMs can immediately be annotated with constraints such as multiplicities at the targets of relation types. A 1...1 multiplicity at B, for example, indicates that a message of type A always causes a message of type B. These multiplicities are written with three dots, since they have asynchronous semantics as will be explained in Section 7.3.

A message type can have many ingoing edges as well, coming from all those transactions that can produce such a message. At the source of each edge, a composition diamond is implied by the above definition of transactions, since every message was produced by exactly one transaction. These diamonds are shown in Figure 8 to illustrate this fact, but they will usually be omitted and assumed implicitly in DTIMs.

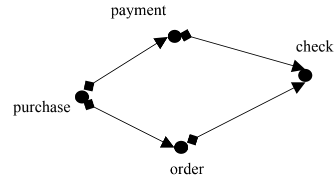

Figure 8. The transitions in DTIMs have implicit composition diamonds at their source.

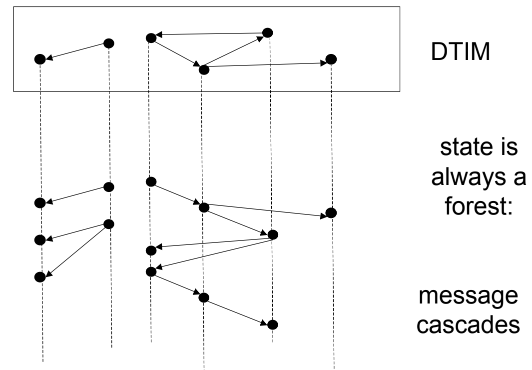

Figure 9. The state of a DTIM is a forest.

This means that the object net over the DTIM is not only cycle-free but in fact it is a forest, as shown in Figure 9, where the DTIM is shown on the top and the state is again shown with swimlanes. A message is processed in a transaction, then the transaction may trigger new messages, and so a *message cascade* is started which is supposed to terminate. A message cascade is always a tree.

7.3 Context of multiplicities

Constraints on DTIMs such as multiplicities can be tied to different checkpoints. For multiplicities this is necessary to accommodate the fact that not all the messages produced during the operation of a system are processed at once but in a consecutive way. To state this in intuitive terms: asynchronous messages call for asynchronous multiplicities. We can formalize this the following way.

A multiplicity tied to the transaction boundaries is valid after each transaction. Such transactional multiplicities are depicted with two dots between the upper and lower bounds. But for the multiplicities given at the targets of DTIM-edges we have the following situation. Mostly, no message at the target of the edge is produced in the same transaction in which the source was produced, therefore the transactional lower multiplicity is zero. However, in DTIMs the processing of messages in subsequent transactions is guaranteed,

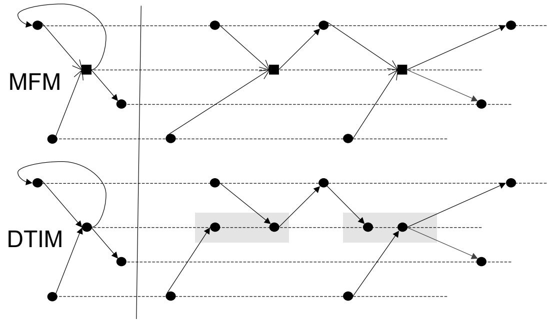

Figure 10. An MFM and a DTIM implementing the former.

and in many cases we know that one subsequent message must be produced later. Here we also want to use multiplicities – we call them *DTIM multiplicities* – to model this knowledge. Since these multiplicities have a different meaning, they are shown with three dots. This distinction is actually only necessary for the lower multiplicity. In DTIMS, these multiplicities typically refer to the end of message cascades. The lower DTIM multiplicities define conditions that must hold before the cascade can terminate, and the upper DTIM multiplicities define limits for the message cascade, although they in general do not suffice to ensure finiteness of the cascade.

7.4 Memo flow and data type interchange

A memo flow model can be implemented by an isomorphic DTIM. An example is given in Figure 10. The DTIM is obtained by removing the distinction between actions and memos. The semantics of the DTIM that is implementing a MFM are the following. If we consider one message type in the DTIM that is implementing an action (in the example there is a single action), then the transaction for this message type has to have the following specification: Only if several instances of this message type have been received, one from each memo required for this action, then the last of these instances sends out all the messages for that action. If we consider the example, then we see that the memo flow model abstracts from the mechanics of data interchange. In the state of the memo flow, the two instances of the action type have an isomorphic star of ingoing edges. In the DTIM state, however, it is denoted that the messages have been received each time in a different order. Therefore the MFM offers a further abstraction that can be helpful but hides the actual operational history, how the memo flow was executed. This is why we call the MFM a computation-independent model.

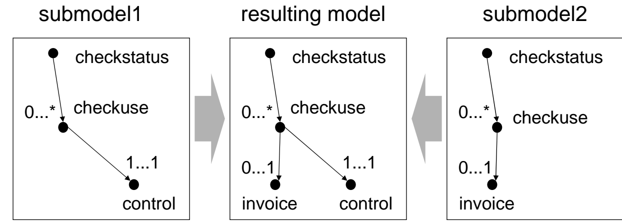

Figure 11. Model Decomposition

7.5 Model decomposition

For the technology-independent models presented here we use the model composition mechanism from form-oriented analysis. Models are conceived as a set of model elements and composition is the set union of these sets. As a consequence, we can compose models that contain partly the same and partly different elements as shown in Figure 11.

In this way we can use one DTIM L to model a subsystem in another DTIM R. We can form larger diagrams, by modeling subsystems and connecting them with a diagram on a higher level. A further discussion of this can be found in [13].

8 Advantages of using core data models

The fact that our models are still core data models has several advantages. This opens up the possibility to create the presented models with many data modeling tools. Core data models are naturally a minimal functionality supported by a majority of modeling tools. Therefore, all these modeling tools can be used to model this type of process model. For example, many tools for UML class diagrams or for ER diagrams can be used. Secondly, because our models are core data models, we can use multiplicities to express integrity constraints on process execution, as seen in our examples. Thirdly the integration with other parts of the data model is immediately possible as shown in Figure 12. The type *shipment* is an entity from the datamodel. The model expresses that for every shipment received by the organization running this system there has to be one or more notification messages sent.

In general, a data modeling tool might not enforce the semantic constraints of our models. For example, it might not enforce the defining constraints on time relation types, and specifically it might not enforce the immutability of memos. In some cases it might even be necessary to rely on standard role names in order to express the directions of time relation types. Also, the well-formedness condition on MFMs, namely that they must be bipartite, will often be not automatically enforceable. Nevertheless, the semantics

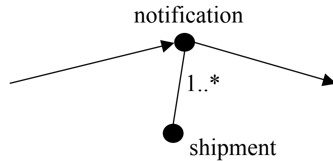

Figure 12. DTIMs can be combined with data models

presented here ensure that such a tool use is more than the trivial option of using a core data modeling tool to draw arbitrary labeled graphs.

9 Conclusion

We need a platform-independent way to describe systems that must fulfil stringent requirements. There is no single definition of the boundary between a computation-independent model and a platform-independent model, despite all strong opinions about these notions. This is aggravated by the fact that what appears to one person as a formal notation for a PIM may appear to someone else as just another data format and hence a proprietary platform. Therefore the main focus should be to ensure that CIMs are on a higher abstraction level as PIMs, and PIMs are not obviously tied to a particular technology. Message-based architectures can be set up with a plethora of technologies. The high-level design of such systems is, however, often very similar. In this paper we have presented computation-independent models and matching platform-independent models that fit well to current state-of-the-art architectures such as enterprise service bus technologies. We have presented semantics for these models by defining them as core data models. This gives us a parsimonious yet powerful modeling approach for process-like business constraints that is easy to integrate with the information model of typical enterprise applications.

References

[1] ITU-T Rec. X.902 — ISO/IEC 10746-2. Open distributed processing - reference model - part 2: Foundations, 1996.

[2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[3] Peter Buneman. Semistructured data. In *PODS '97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121. ACM Press, 1997.

[4] Christoph Bussler, Dieter Fensel, and Alexander Maedche. A conceptual architecture for semantic web enabled web services. *SIGMOD Rec.*, 31(4):24–29, 2002.

[5] Barry Dowdeswell and Christof Lutteroth. A message exchange architecture for modern e-commerce. In Dirk Draheim and Gerald Weber, editors, *TEAA*, volume 3888 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2005.

[6] Dirk Draheim and Gerald Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.

[7] Marco Eichelberg, Thomas Aden, and Jörg Riesmeier. A survey and analysis of electronic healthcare record standards. *ACM Computing Surveys*, 2005.

[8] Margaret A. Emmelhainz. *EDI: Total Management Guide*. John Wiley & Sons, 1992.

[9] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[10] Paul Kimberley. *Electronic Data Interchange*. McGraw Hill, 1991.

[11] Min Luo, Benjamin Goldshlager, and Liang-Jie (LJ) Zhang. Designing and implementing enterprise service bus (esb) and soa solutions. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, page .14, Washington, DC, USA, 2005. IEEE Computer Society.

[12] D. Moberg and R. Drummond. MIME-Based Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2). RFC 4130 (Proposed Standard), July 2005.

[13] Gerald Weber. A platform-independent approach for auditing information systems. In *HDKM '08: Proceedings of the second Australasian workshop on Health data and knowledge management*, pages 65–73, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[14] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture*. Prentice Hall PTR, 2005.

[15] Han Zhang, Gerald Weber, William Zhu, and Clark Thomborson. B2b e-commerce security modeling: A case study. In *Computational Intelligence and Security, 2006 International Conference on*, pages 1549–1554, 2006.