A Platform-Independent Approach for Auditing Information Systems

Gerald Weber

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

Abstract

Information systems in several application domains have to fulfil particularly stringent requirements, first of all concerning privacy, but then also concerning the ability to audit the use of data in hindsight. For databases as a key component of such systems, the concept of hippocratic databases was proposed (Agrawal et al. 2002). These databases are targeted at privacy-intensive applications including healthcare applications. Hippocratic databases enable active enforcement of privacy policies, as well as audits of compliance. We present here a framework that allows us to audit the data that was actually presented. In a model-driven approach, platform-independent models support reuse and are translated into platform dependent models. We present here a platform-independent model for auditing information systems. It is based on a message-based system viewpoint that allows us to discuss aspects of a service-oriented architecture on a high-level analysis and design level. This method shows how we can use a protocol of all ingoing and outgoing messages as an audit trail for the system.

1 Introduction

In this paper we discuss a platform-independent model for message-based communication of information systems. We will see how the message model of this approach fosters our understanding of the system and enables powerful data analysis. In particular it will enable us to perform audits of the system usage. Information systems in several application domains have to fulfil particularly stringent requirements, first of all concerning privacy, but then also concerning the ability to audit the use of data in hindsight. There are many application areas, in particular in the healthcare sector, where there is an increasing demand for reliable auditing features, partly motivated by policies. In practice such policies often already have connections to message-based concepts such as EDI. For databases as a key component of such systems, the concept of hippocratic databases was proposed (Agrawal et al. 2002). These databases are targeted at privacy-intensive applications including healthcare applications. Hippocratic databases provide support for active enforcement of privacy policies, as well as for audits of compliance with these policies. The compliance audit is based on database accesses. If we want to interpret these database accesses, however, we have to make some assumptions on the information system that processes the data before the presentation to the user. In contrast, we present here a framework that allows us to audit the data that was actually presented. Our framework allows us to audit end-user interaction with the system, as well as interaction via service-oriented interfaces.

We use a form-oriented system model (Draheim & Weber 2004), because in this way the message-based audit-data can be obtained in a structured data format that can be easily mapped to relational data. This makes it possible to store such data in a meaningful way directly in a data analysis component such as a data warehouse or in Hippocratic databases in order to use the advantages of this technology not only to the primary data, but also to the audit data, such as limited disclosure (LeFevre et al. 2004).

This paper is first of all focusing on presenting the underlying system model, the paper naturally leaves the discussion of detailed audit functions to further presentations. This system model can not only be applied to human-computer interaction, but also to the boundaries of discernible subsystems of an information system. For the latter purpose we introduce a platform-independent model for message-based communication between automated systems. One key idea of a model-driven approach is to use platform-independent models in order to enable reuse to translate them into platform-dependent models. 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 web-services roll-out is an implementation technology and not an architecture.

We discuss here a platform-independent model that has been successfully applied in industry projects. Today's enterprise computing projects are covering areas as diverse as healthcare (Eichelberg et al. 2005) and e-commerce (Zhang et al. 2006). In such projects, a plethora of different message-based technologies is used; take just e-commerce with classical EDI (Kimberley 1991) and AS2 (D. Moberg and R. Drummond 2005), a novel e-commerce standard that is similar but different from web-services (Bussler et al. 2002, Weerawarana et al. 2005). We capture our modeling approaches as viewpoints in the tradition of Open Distributed Processing (ODP) (ISO/IEC 1995, Farooqui et al. 1995).

The main line of argument of this paper is as follows. In Section 2 we reflect on the importance of message-based communication for enterprise computing. In Section 3 we motivate our approach with the viewpoint approach of ODP and discuss a key modeling principle, namely the heavy use of immutable datatypes. In this paper we focus on statically typed system models. The methodology here can also be generalized to untyped or dynamically typed system models, but many parts of such generalizations are rather straightforward, moreover they incur the usual loss of static expressibility and are of lesser interest for the comparison with other formalisms. The message-based viewpoint presented in Section 4 captures the high-level architecture of state-of-the-art systems, particularly enterprise service bus architectures. This viewpoint gives rise to our platform-independent high-level design models; they are called Data Type Interchange Models (DTIMs) and are designed to match these modern message-based system architectures. In Section 5 we explain how this gives a message-based model of human interaction with the system. In Section 8 we discuss the system view of form-oriented analysis. In Section 7 we discuss how this interface model allows us to define system viewpoints in the ODP terminology; these viewpoints in turn allow us to discuss important features that we expect from an auditable information system.

2 Connectivity of Enterprise Systems

The automated communication between systems is one of the cornerstones of an efficient IT infrastructure within or between organizations. The challenges here are not just the different technologies used by different organizations and the agreement on the semantics of the exchanged messages. A further important issue is related to system stability. An organization participating in automated communication wants to make sure that the use of the automated interface does not endanger the stability of its own system. But also within a single organization very similar stability demands for system communication appear, since the desire is to decouple the systems with respect to failures so that no single event can endanger the whole IT infrastructure.

2.1 Message-Based Middleware

An established best practice for the implementation of data interchange between different system units is the use of message-based middleware. In using message-based middleware, systems can send information asynchronously to other system parts. The delivery can be configured to follow different quality of service (QoS) levels. A particular high-quality service can be obtained by the use of persistent messaging. A crucial element of the message-based infrastructure and a key difference to simple remote method invocation is the existence of queues which store the messages until they are retrieved by the receiver. The queues fulfill two intertwined purposes. First they provide a buffer for bridging temporal load peaks; of course this means that the queued requests experience a delay in response time. Secondly they provide a means of secure intermediate storage; in the case of persistent messaging this is a transactionally safe storage. Today, persistent messaging is done through high-quality Enterprise Service Buses (ESBs), as shown in Figure 1. Asynchronous messaging services can be used to build synchronous request/response services on top of them. The underlying principle is ubiquitous in computing and also found where synchronous services are built on top of IP, which is itself asynchronous. We recall that the transaction service enables consistent multi-user business logic on shared data. Persistent messaging now is attractive, since it makes it possible to combine independent transactional systems in a transactionally safe manner, without creating the considerable overhead of a distributed transactions framework. The message-based paradigm is the foundation of our network model. The consistent use of this paradigm not only provides a simple and adequate model for specification purposes, but also ensures that the specification is directly translatable in many current architectures, including persistent messaging and Web services. Our message-based network paradigm is part of the high-level programming paradigm.

2.2 Electronic Data Interchange

Message-based data interchange is used in many mature technologies. EDI is an implementation technique for business message interchange (Emmelhainz 1992). EDI allows communication with high QoS, which can furthermore be customized. The interchange is traditionally on dedicated networks, so–called value–added networks. EDI furthermore defines industry-wide message types, which can be seen as one of the most important examples of domain models, i.e., common models for an application domain. A system based on EDI can easily be modeled with form-oriented techniques, and the form-oriented description can be seamlessly mapped to a design for a system communicating with EDI.

2.3 Web Services

Web services (Alonso et al. 2004) are an implementation technique for message communication. They use XML as a semistructured format (Buneman 1997) for messages. Notations for distributed systems based on Web services are called Web service orchestration languages such as BPEL (TC 2007). 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., about 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.

3 The Memo Based 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." (ISO/IEC 1995). We will propose our own viewpoints, some of them are related to the enterprise viewpoint of ODP; each of our viewpoints gives rise to a particular important modeling concept.

Our first viewpoint is the *memo-based viewpoint* and it captures the requirements of a rigorous document quality management as it is current best practice. The central architectural feature of this viewpoint is that the system model must contain a single space of memorandums, memos for short. In this system view, all IT-related business activities proceed by creating documents, namely the memos, typically based on earlier documents. Memos are only considered created at the moment the user or the system signs them, and after this point in time they are immutable in a truly persistent way and remain in the system state. Hence the memo viewpoint models archiving on an analysis level as the repository of memos. User generated as well as relevant system generated memos must be kept. We want to model memos in the data model. Hence memos must be modeled as immutable composite data objects.

In a stronger version, the *fully* memo based viewpoint, it is established that the whole system state can be derived from this repository of memos, and

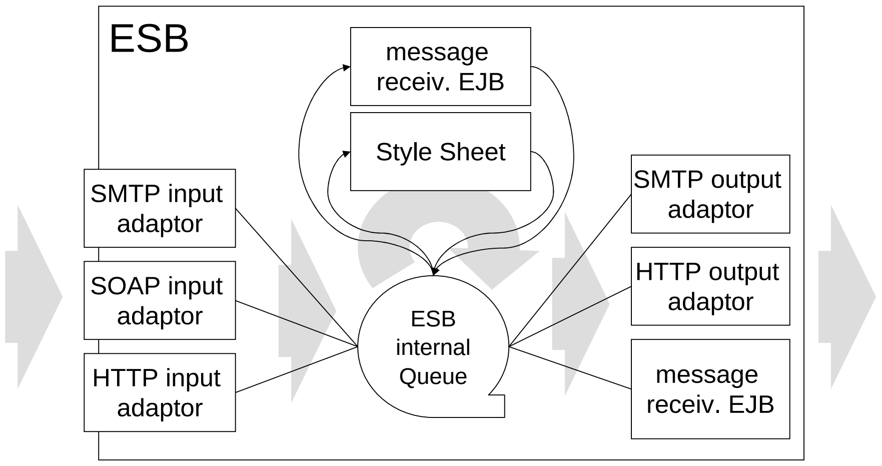

Figure 1: The high level design expressed in DTIMs fits to Enterprise Service Bus (ESB) Architectures.

hence this repository *is* the system state. The classical changeable system state can then be seen as a mere view on this repository as it would be possible using database view technology. This state can still be effectively updateable, but the update commands would be stated as memos, and the definition of the current state as a view has to be such that an inserted update memo has the desired effect. Take a collection of accounts as an example. The update memos could be credit/debit memos for accounts. The table of current balances of the accounts is a view that is defined by taking the sum over all credit/debit memos for each account; in this way the insertion of such a memo leads to the desired effect. A more detailed discussion of this view will be given in Section 8.

The memo-based viewpoint is not supposed to fit to all computer applications; for example real-time control applications are different. But it is a powerful tool for administrative applications.

The memo-based viewpoint makes clear that immutable memos are capable of maintaining a model of the system development, because the memos reference back to earlier memos; this will be elaborated in the Memo Flow Model.

As such, it is again an enabling viewpoint for later process models. Memos are a precursor to messages, but in memos there is no emphasis on them being sent. The memo-based viewpoint is a slight alteration of a message-based viewpoint: The aspect of transporting information is not yet emphasised, and persistence is emphasized more.

4 The Message-Based System Viewpoint

We will use memos to describe messages. Messages are therefore persistent and kept in a repository, the message model. For the sake of regularity, in this viewpoint all memos are considered to be messages, therefore it is called the *message-based* viewpoint. Furthermore we assume the messages to be strongly typed. Each model in this viewpoint is called a *data type interchange model*, DTIM for short. DTIMs are high-level design models for message-based communication. In this viewpoint we will capture the architecture of state-of-the-art message-based systems. They fit well for example to enterprise service bus architectures (Luo et al. 2005), but also to similar systems used in electronic data interchange (Dowdeswell & Lutteroth 2005).

The core architectural feature of such systems is that the business logic is comprised 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. In Figure 1 two components are shown as transformations that take messages and produce again other messages.

4.1 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. 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, the unit system offers a set of *transactions* it can perform. 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. A transaction takes a message and produces a set of new messages. The transition function can now be written as:

$$\begin{aligned}\texttt{transaction}_a\texttt{:}\ \ & \texttt{message}_a \times \texttt{state} \rightarrow \texttt{state}\\ & \times\ \texttt{collectionOf(message}_{a_1}\texttt{)}\\ & \times\ \texttt{collectionOf(message}_{a_2}\texttt{)}\\ & \ldots\\ & \times\ \texttt{collectionOf(message}_{a_{n_a}}\texttt{)}\end{aligned}$$

$$\begin{aligned}\texttt{transaction}_b\texttt{:}\ \ & \texttt{message}_b \times \texttt{state} \rightarrow \texttt{state}\\ & \times\ \texttt{collectionOf(message}_{b_1}\texttt{)}\\ & \times\ \texttt{collectionOf(message}_{b_2}\texttt{)}\\ & \ldots\\ & \times\ \texttt{collectionOf(message}_{b_{n_b}}\texttt{)}\end{aligned}$$

$$\vdots$$

We can therefore represent the transactions by their message types. We call the message type the *superparameter* of the transaction. Each such transaction may represent one deployed transformation component from Figure 1. Message-based components are an old type of components, known from classical transaction monitors. They are crucially important for a workable enterprise platform, and it was a major

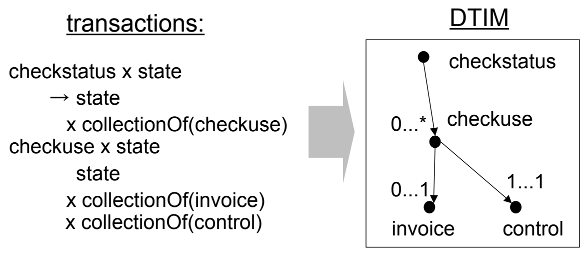

Figure 2: DTIMs fit to the typed automaton model.

drawback for the early Java enterprise platform not to have message-driven ESBs; 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, and 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 resemblance to the coordination language Linda (Gelernter 1985), but in practice the central requirements for users are nonfunctional requirements, chiefly persistence of the messages. There is also a direct correspondence to 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.

4.2 Data Type Interchange Models

The Data Type Interchange Models (DTIMs) that we are going to introduce are first of all a natural translation of the above mathematical definition into a data model. We will use the parsimonious data model from form-oriented analysis (Draheim & Weber 2004). The DTIM in its elementary form contains as entity types (depicted as nodes) just such message types that represent transactions. This means that we depict such message types where each new instance triggers an action, and this action may result in new messages. The messages are immutable types.

An elementary DTIM is a directed graph with message types as nodes. If the transaction for message type A may send messages of type B, then the DTIM contains a directed edge from A to B. The connection between the mathematical notation for a single transaction and the node in the DTIM is shown in Figure 2. DTIMs are just data models, the nodes are entity types and the edges are relation types between them. The direction on the edges implies an order in time of the insertion of messages, from earlier messages to later ones. 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 4.5. 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 3 to illustrate this fact, but they will usually be omitted and assumed implicitly in DTIMs. A message type can have many ingoing edges as well, coming from all those transactions that can produce such a message. This means that the object net over the DTIM is not only cycle

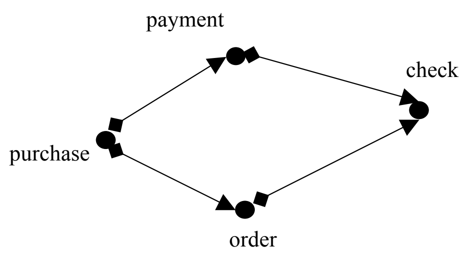

Figure 3: The transitions in DTIMs have implicit composition diamonds at their source.

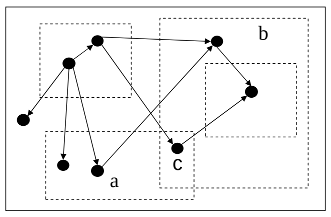

Figure 4: Subsystems in a DTIM

free but in fact it is a forest.

4.3 The Distribution Viewpoint

So far, elementary DTIMs model message-based system architectures and view distribution as *transparent*, meaning that distribution is not depicted. In the distribution viewpoint, we want to demarcate different distributed systems.

We introduce subsystem types, depicted by dashed boxes around message types. Each such box is a subsystem type. Such a subsystem type can have many instances. Each message, being an instance of a message type, must belong to one subsystem instance. Boxes can overlap, however, or message types can be depicted several times, in different subsystems. For example, in Figure 4, the subsystems a and b overlap and have message type c in common. This means that a message instance of type c can be either in subsystem a or in b, but not in both. We therefore have the notion of a unit system type and a single unit system of this type. Several servers that run the same software would be modeled as several instances of the same unit system type. If a subsystem type is for example modeling a supply chain management system, then each instance would model a different company using this software and the extension (set of all instances) of the subsystem type is the set of all banks that use the same enterprise software. If another part of the DTIM models a supplier and this supplier sends a message to the supply chain system, then we see that the DTIM does not express the interconnection of the system instances; rather it models the fundamental capabilities of the software: It shows that a supplier with a certain software can connect to one instance of the supply chain management product, then this supplier can in principle connect to all systems of this make. Usually we expect that the subsystem boundaries are boundaries for the access to data that the actions can take in their operations.

4.4 Compositional Properties of Data Type Interchange Models

We use for the platform-independent models presented here the model composition mechanism from

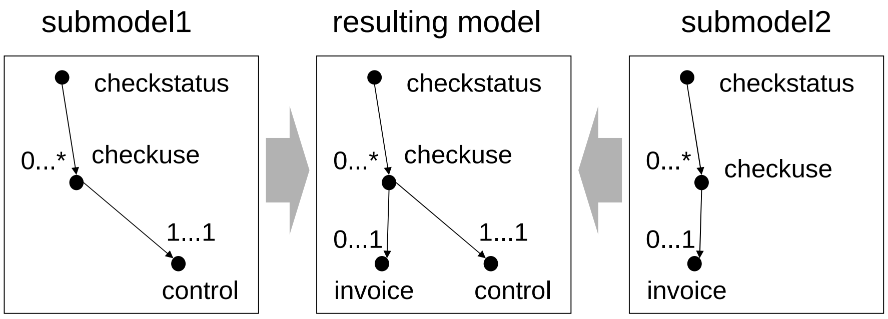

Figure 5: Model Decomposition

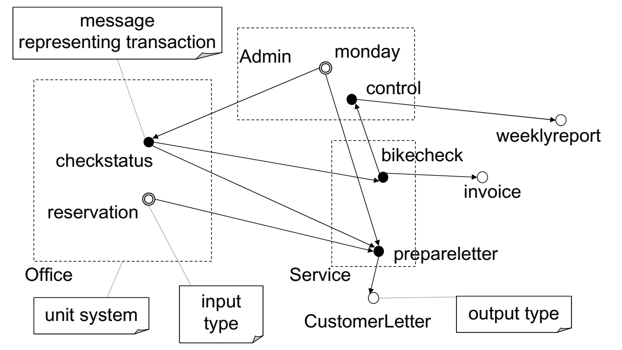

Figure 6: An example data type interchange diagram showing all the notational elements

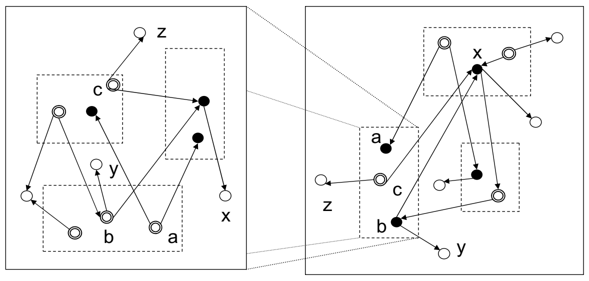

Figure 7: Similar to functional decomposition, a model subsystem in one DTIM can be defined by a whole sub-DTIM

form-oriented analysis. the point of this mechanism is that models are conceived as a set of model elements and composition is the set union of these sets. As a consequence, some models that contain partly the same, partly different elements, can be composed, as shown in figure 5.

In order to indicate that the system is producing output to the environment, it has to have message types which do not represent specified transactions, but outgoing messages. These message types form the *output collection*, each one called an *output type*. In the following we also want to have the possibility to explicitly define which messages can be received from the environment; we call them *input types*, together forming the *input collection*. Each choice of messages from the model is valid as the input collection. By not including a message in this collection we express that this message does not relate to a publicly available service of the distributed system. This is therefore a visibility stereotype. An example showing the notational elements is shown in Figure 6.

The input collection is helpful for an analysis in the style of dead code analysis. Transactions with superparameters not reached from the input collections can never be called. In Figure 6 the transaction order is such an example.

The whole distributed system expressed by a DTIM can itself be conceived as having a single interface. For each input method we choose those messages from the output list which are reachable in the directed graph. In this way, the whole system can be considered as a single subsystem in a larger system.

In this way we can use one DTIM L as a model subsystem in another DTIM R. This is shown in Figure 7. The relation between the two diagrams shows that this is a compositional concept. We can form larger diagrams, by modeling subsystems and connecting them with a diagram on a higher level. This is similar to functional composition. The subsystem can be said to be called over its interface by the higher level system. It is important to add that the semantics behind this composition notion is just the concept of model composition by set union that was explained earlier. Concerning the outgoing arrows of a message type, we introduce here more general use of DTIMs, because in this diagram, the outgoing edges of one action do not have to be produced in the same transaction; they can have been produced in a later transaction that was *caused* by the first transaction.

4.5 Context of Multiplicities

Constraints on DTIMs such as multiplicities can be tied to different checkpoints. For multiplicities this is necessary to accommodate for 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 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 multiplicities are depicted with two dots between upper and lower bound. But the multiplicities given at the targets of DTIM-edges – we call them *DTIM multiplicities* – have a different meaning, and are therefore shown with three dots. To understand the motivation we discuss the effect of a message. The 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. The lower DTIM multiplicities define conditions that must hold before the cascade terminates, and the upper DTIM multiplicities define limits for the message cascade.

5 Application to Form-Oriented Interface Modeling

DTIMs can be used to specify interaction with a browser. We compare here formcharts, the dedicated user interface models of form-oriented analysis (Draheim & Weber 2004), with an equivalent DTIM. The terminal client where the dialogue according to the formchart takes place is usually one subsystem. All client sessions are instances of this unit system.

Within a server action modeled in a DTIM it is possible to query a remote system and to forward the response to the user. Take a look at the example in Fig. 8. In this example we assume that the user performs a login at a subsystem of the IT infrastructure, but the authentication service is on a remote system. The example shows a formchart with one server action LoginForm. The two solutions below show two different specifications of this formchart cutout with DTIMs. The interface of the remote customer management determines which alternative is the right one.

All client pages are in the leftmost unit system, the form-oriented client. The transitions from the leftmost unit system to the unit system in the middle represent all page/server transitions in the formchart. The node LoginForm in the IT subsystem represents the actual message type LoginForm from the user message model. The submitted form goes to this transaction. The subsystem queries an external customer management system about the authentication. This system answers to a receiver transaction. There are two natural variants for the interface of the authentication service. In solution A it sends back one message type containing the statement on the validity of the user request. In solution B it sends an error message in the case of an invalid request. The subsystem then has to offer two receivers for the respective message. The transition between Login and LoginForm is again another generalisation of the use of DTIMs. The transition in this case takes place after the user has viewed the login form and triggered a new page change. In this way the DTIM is even used to model processes that include human intervention.

6 The Action Viewpoint

An often discussed phenomenon is synchronization between message streams. Not every incoming message must immediately lead to an action. Assume, we need a quote and an approval in order to confirm a travel booking. Then only if both messages are arrived, an action is performed. We therefore want to define a viewpoint that expresses this abstraction, the *action viewpoint*. Actions that are performed get an own identity, and they can result in several memos 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 9. The two partitions are actions and memos. The actions themselves do not hold data, but they help in defining a superparameter; the action is linked to all the memos that are input to the action, and we consider the composite of all these memos the superparameter. The action viewpoint is a computation-independent model (CIM). It can be implemented with DTIMs.

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

Often, with respect to transactions, the actions are executed detached; that means: in a new transaction after the trigger conditions are met; the trigger conditions then hold already before the action is performed. In principle, one message insert can trigger multiple actions, and these actions do not have to be in one transaction.

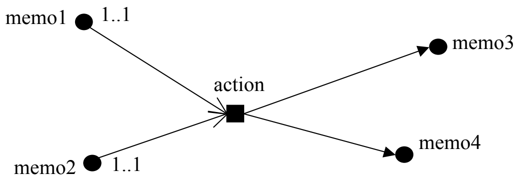

Figure 9: A Message Flow Model

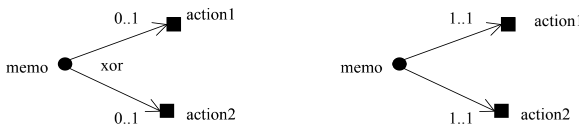

Figure 10: Nondeterminism in a Message Flow Model

6.1 Nondeterminism and the Action Viewpoint

The action viewpoint is similar to coloured petri nets since it is bipartite. In contrast to Petri nets however, there is not necessarily a consumption of messages by the successor messages. This becomes obvious if we consider nondeterminism in examples. In the two small MFMs in Figure 10, 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). That could express nondeterminism, but not necessarily; this should come as no surprise, since nondeterminism could be often a lack of information, so if other parts specify when each alternative will be used, the nondeterminism disappears. This is a typical instance of the fact that nondeterministic programs are often simplifications of deterministic programs, as it was for example used in the JSPick type system for loops and if statements (Draheim et al. 2003).

On the right hand side however, the message may give rise to two actions because of the multiplicities. Having an upper multiplicity at every action reachable from a memo, like in Figure 10 is the usual case. This is important to make sure that the trigger is executed only once, without referring to a very mechanical operational semantics of trigger execution. The lower multiplicity zero is necessary in many cases, since actions are executed detached, and it is hard to give any guarantee on when the message will be executed. The main advantage over more sophisticated Petri Net variants is here that we use only the most basic core of data modeling; that is, entity types, relation types and multiplicities, nothing more.

The bipartiteness in the action model is different from the bipartiteness of a formchart: The formchart is a further partitioning of the memos into two partitions; the actions are not modeled in the form chart, because the modeling of the actions is trivial in the case of the form chart. This is because in the classical form-oriented model, every action is triggered by a single message; in other words there is no synchronisation going on. The actions are intermediate states between the formchart states, thus representing transitions.

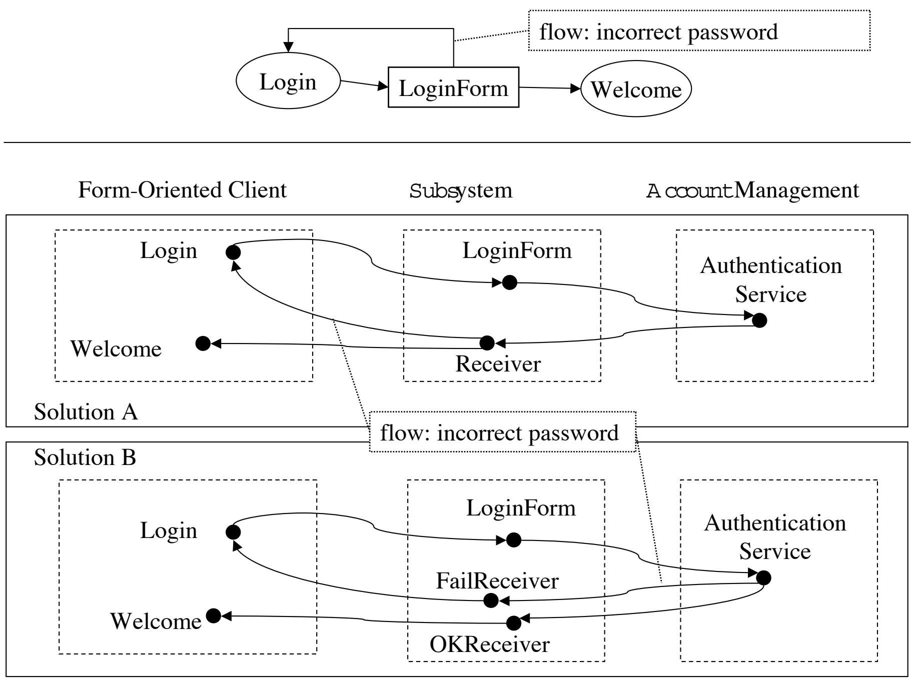

Figure 8: Example DTIMs describing how the login dialogue accesses a remote authentication service

7 Viewpoints that Support Auditability

Now we are in a position to use the model obtained so far in order to provide a high-fidelity audit track of the user interaction. In the last section we have established a purely message-based model for system interaction. In this section we will present how this model should be applied in order to make sure that the recorded messages represent faithfully the system interaction. Again we state this as viewpoints in the style of ODP.

7.1 The User Memo Viewpoint

In this viewpoint, user created memos are strictly distinguished from all other, system created messages.

We extend it to the *Signed User Memo Viewpoint* with the following main semantic model: Users prepare and sign memos, wherein signing is understood in a very general sense — hitting a confirm button after explicit exhortation to check carefully counts as signing; this will be discussed later. If a user shall sign a memo, then the memo must contain a complex information that makes sense. Hence the memos must be semantic units that create a level of hierarchy in a hierarchical data model. Some memos are irreducible units on their semantic layer; that means, decomposition down to smaller units is not possible without loss of meaning (while they can probably be always composed, creating what is called a junctim). Wherever sensitive data is involved, the practitioner must be aware of the significance of submitting a certain report. This viewpoint clarifies that if we later want to audit the submission of a certain primitive value, we need to store sufficient context information; this is the memo.

The *system output viewpoint* is complementary to the user memo viewpoint. It separates out those messages that have been presented to the user (as opposed to having been just put into a mailbox). Here, however, there are some issues in defining which information has been shown to the user. In principle, even a page that has to be scrolled is not an output in itself, but only a queryable result to begin with, the query being the scrolling option. Therefore, especially for critical systems, we will have to define a *screen output viewpoint.* In this viewpoint, a faithful model of the information as it was presented to the user does exist, and it is available for audits.

7.2 The Plain Data Memo Viewpoint

The signed user memo viewpoint requires that the information was shown to the user and recognized. This poses interesting questions: Which formatting is permitted, for instance regarding font size, color, reflowing of text. How is the message identified? Which framing is necessary? For critical information we should avoid embellishment. Such information should be presented as plain data. Plain data is not plain text, it is a successor that it is similarly bare-metal, but problem-defined, not implementation-defined as it was the old green screen terminal. All graphical formatting should follow strict guidelines. The use of color must be questioned. Emphasis like bold fonts should be only used if this follows a well known standard in the profession using this system. A user interface element that contains important alerts must be prevented from occlusion, not only from scrolling, but from other windows, minimization etc. Occlusion-free technologies like the generic editor in (Lutteroth 2006) can be used. There is no point in auditing the system output on the GUI level, if there is no inherent guarantee that the user has actually seen the messages that are registered in the audit trail. Information presentation is on/off: Either information is presented, or it is not presented. This is particularly the perspective that a later audit will take in any case. Plain data can be described with data types, and an instance of the data type is sufficient to describe a presented page, since the presentation should be standardized. Moreover, one has to be aware that data presentation always involves the presentation of data *and* metadata. It is a reasonable requirement that a measurement value is presented with the same label at the output, as it was presented at the input. This label is part of the metadata, and in many cases part of the type system. In form-oriented analysis, this uniform presentation is achieved by collecting the metadata in a repository, the shared model.

Pages that present information as well as the forms that are used to enter the data are specified as types of a hierarchical type system. The form is represented by the same type that is used to describe its presentation. The plain data viewpoint would therefore require that in general the screen presentation is always the same for the same piece of data. In principle only the access rights change (whether a piece of data can be read or written).

7.3 User Notification Viewpoint

in the *User Notification Viewpoint* we emphasize that important events have to be presented such that their arrival is recognized, such as critical information that must be read in full by the user. This concept belongs under the user memo viewpoint, since the natural idea is that the user has to sign them off with a short, generic I-have-read message. Furthermore the sign-off should follow the following pattern that is of more general relevance.

The confirm pattern is a proposal how to standardize signing and how to make it recognizable for the user. In the confirm pattern, all binding signing operations happen on forms that have only immutable content and have only a confirm and an exit button. The exit button is not a reject button, in case it is a modal dialogue. This means, the user is not coerced to either accept right now or lose the offer, because this would prevent the user from doublecheck other data in the system, or, more general, to do higher prioritized tasks. If it is not a modal dialogue, then there is not a necessity to require this exit semantics. There is only one possible change the system can do without being prompted by the user, and that is to invalidate the offer. An additional requirement could be that minimum 'offer valid until' time is shown, and the offer is only invalidated after that time. Beyond that, the user confirm viewpoint requires a receipt for the user, so that the user knows that his confirmation was indeed recorded.

8 Form-Oriented Models

The system models given so far create the basis for the modeling methodology that we call form-oriented analysis (Draheim & Weber 2004). There it was argued that we can capture information systems in certain highly standardized models that we call normal forms. These normal forms are different from the database theory notion of normal forms. Our normal forms will allow us in turn to audit such systems. They all have the term warehouse in their name to indicate that they all support inserts, but no updates. There are three normal forms that we usually consider:

- The history warehouse. Every state the system assumed at a certain point in time is kept as part of the immutable system state. This system view is therefore adding a time dimension to the whole datamodel.
- The log warehouse. The log of updates is kept as a queryable data model. First we assume that the log is kept in addition to the current (updateable) state. The log is a redo log, and this means by definition that the information in this log is sufficient to reconstruct the state at each point in time. The state thus is revealed as being a materialized view on the redo log.
- The message warehouse. In the message warehouse, every incoming and outgoing message is stored in the system state. This model stores the messages as they appear at the system boundary.

The message warehouse stands in direct relationship to the fully memo-based viewpoint. These normal forms enable us to give a rewording of the tenet of the fully memo-based viewpoint. The idea behind this viewpoint is that all three normal forms are in fact equivalent under certain assumptions; they can be derived from each other. Therefore we want to make these further assumptions about the business logic explicit. In order to assume that the log warehouse is a view on the message warehouse, we have to assume that the business logic makes any nondeterminism known to the system clients. This means that the same input message always leads to the same updates, or any nondeterminism taking place has to be reported *in the same transaction* in some kind of output (which could, however, be an administrative message inbox). In order for the message warehouse to be a view on the log warehouse, we have to assume that the system stores all input in some manner, i.e. it does not discard any input field. With very few exceptions (repeated password in a password change dialogue) this is a natural assumption anyway. On the other hand this points out that it is a good practice for a system to keep a direct record of all input: it is indeed a strange procedure if we shred each input form and distribute it onto many tables of the data model without keeping a direct record.

The normal forms are important for us because they deliver the audit trails that we need in order to assess the system behavior. Hippocratic databases use in our terminology the log warehouse for auditing purposes. We however will put emphasis on the message warehouse, because this keeps track of all the system communication and hence can be used as the audit trail. We argue that it is moreover the more natural audit trail, because it keeps more direct evidence of the system interaction, especially if embellishment is kept to a minimum, which is a good practice in critical systems as we will argue later. The log warehouse in contrast has to be interpreted and the actual user activity has to be inferred. The message warehouse is interesting in that it connects data modeling with human computer interaction: all interactions become part of the data model.

9 Discussion

The viewpoints give us a platform-independent list of requirements. They particularly help us to use the message-warehouse as a faithful history of the system usage for further audits. Compliance audits enable an after-the-fact analysis whether privacy policies have been observed (Johnson & Grandison 2007). Our system model keeps a log of every interaction with the critical system. This includes also the information about the conversational sequence in which the messages are produced. The amount of data presented, especially if reports are involved, may become large. However, in sensitive information systems it is reasonable to assume that casual browsing is discouraged. Futhermore the audit trail should describe faithfully what was actually presented to the user (following the screen output viewpoint). In the case of a report this is therefore not necessarily the whole report, but only the pages that were seen by the user. The audit process in our proposal has access to a description what information was actually presented to the user. In principle, it might not be automatically possible to reconnect the data presented to the database entries; this is so because the intervening information system, as a potentially universal system, can arbitrarily process the data. In the simplest such case the label of the data might be changed (column in the database has an abbreviation as a name, form on the screen presents the long name). It is therefore indeed necessary to test the audit process for every type of screen or every outgoing message that the system generates. This is another example where the system model presented in form-oriented analysis can help: It clarifies that a reasonable system can be built with only a finite number of screen types and is therefore amenable to such a check. Furthermore this clarifies why the use of a shared model as explained in the plain-data viewpoint is crucial not only from the standpoint of user understanding of the system, but also from the standpoint of auditing the system. Our approach offers a further possibility to enhance privacy: If the concept of a shared model is used consistently, then we can envisage a purely formal audit trail, in which for one message of a part of this message only the type from the shared model is recorded, and the sensitive data is not replicated in the audit trail.

10 Conclusion

We need a platform-independent way to describe systems that must fulfil stringent requirements. Here we focused on an architecture enabling auditability. We used a message-based approach since the messages are the building blocks of our audit trail. 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 presented a platform-independent model that fits well to current state of the art architectures such as enterprise service bus technologies. We focused on making systems auditable, not on discussing the audit process itself. We have argued that the audit process should have access to the data in a way that faithfully represents the way the data was presented as well as the way the data was entered. At the same time this faithful way must be on the level of the data itself, i.e. it must live on the abstraction level of data types. In the future, such a system model can be used to provide standardized input for audit systems based on standard technologies such as data warehousing and data mining.

References

Agrawal, R., Kiernan, J., Srikant, R. & Xu, Y. (2002), Hippocratic databases, *in* 'VLDB', Morgan Kaufmann, pp. 143–154.

Alonso, G., Casati, F., Kuno, H. & Machiraju, V. (2004), *Web Services: Concepts, Architecture and Applications*, Springer Verlag.

Buneman, P. (1997), Semistructured data, *in* 'PODS '97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems', ACM Press, New York, NY, USA, pp. 117–121.

Bussler, C., Fensel, D. & Maedche, A. (2002), 'A conceptual architecture for semantic web enabled web services', *SIGMOD Rec.* **31**(4), 24–29.

D. Moberg and R. Drummond (2005), 'RFC4130: MIME-Based Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2)', RFC.

Dowdeswell, B. & Lutteroth, C. (2005), A message exchange architecture for modern e-commerce., *in* D. Draheim & G. Weber, eds, 'TEAA', Vol. 3888 of *Lecture Notes in Computer Science*, Springer, pp. 56–70.

Draheim, D., Fehr, E. & Weber, G. (2003), JSPick - a server pages design recovery, *in* '7th European Conference on Software Maintenance and Reengineering', LNCS, IEEE Press.

Draheim, D. & Weber, G. (2004), *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*, Springer.

Eichelberg, M., Aden, T. & Riesmeier, J. (2005), 'A survey and analysis of electronic healthcare record standards', *ACM Computing Surveys* .

Emmelhainz, M. A. (1992), *EDI: Total Management Guide*, John Wiley & Sons, Inc., New York, NY, USA.

Farooqui, K., Logrippo, L. & de Meer, J. (1995), 'The iso reference model for open distributed processing: an introduction', *Comput. Netw. ISDN Syst.* **27**(8), 1215–1229.

Gelernter, D. (1985), 'Generative communication in linda', *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112.

ISO/IEC (1995), 'Open distributed processing- reference model - part 2: Foundations international standard 10746-2 itu-t recommendation x.902'. **URL:** *citeseer.ist.psu.edu/3915.html*

Johnson, C. & Grandison, T. (2007), 'Compliance with data protection laws using hippocratic database active enforcement and auditing', *IBM Systems Journal* **46**(2).

Kimberley, P. (1991), *Electronic Data Interchange*, McGraw Hill.

LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y. & DeWitt, D. J. (2004), Limiting disclosure in hippocratic databases, *in* M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley & K. B. Schiefer, eds, 'VLDB', Morgan Kaufmann, pp. 108–119.

Luo, M., Goldshlager, B. & Zhang, L.-J. L. (2005), Designing and implementing enterprise service bus (esb) and soa solutions, *in* 'SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing', IEEE Computer Society, Washington, DC, USA, p. .14.

Lutteroth, C. (2006), AP1: A platform for model-based software engineering, *in* D. Draheim & G. Weber, eds, 'TEAA', Vol. 4473 of *Lecture Notes in Computer Science*, Springer, pp. 270–284.

TC, O. W. S. B. P. E. L. W. (2007), 'Web Services Business Process Execution Language (WS-BPEL) Version 2.0'.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T. & Ferguson, D. F. (2005), *Web Services Platform Architecture*, Prentice Hall PTR.

Zhang, H., Weber, G., Zhu, W. & Thomborson, C. (2006), B2b e-commerce security modeling: A case study, *in* 'Computational Intelligence and Security, 2006 International Conference on', pp. 1549–1554.