Application of Information Technology ■

A Practical Approach to Process Support in Health Information Systems

Richard Lenz, DrIng, Thomas Elstner, Dipl.Phys, Hannes Siegele, Klaus A. Kuhn, ProfDrMed

Abstract This article describes the design of a generator tool for rapid application development. The generator tool is an integral part of a healthcare information system, and newly developed applications are embedded into the healthcare information system from the very beginning. The tool-generated applications are based on a document oriented user interaction paradigm. A significant feature is the support of intra- and interdepartmental clinical processes by means of providing document flow between different user groups. For flexible storage of newly developed applications, a generic EAV-type (Entity-Attribute-Value) database schema is used. Important aspects of a consequent implementation, like database representation of structured documents, document flow, versioning, and synchronization are presented. Applications generated by this approach are in routine use in more than 200 hospitals in Germany.

■ **J Am Med Inform Assoc.** 2002;9:571–585. DOI 10.1197/jamia.M1016.

Clinical processes are characterized by a high degree of communication and cooperation among physicians, nurses, and other groups of personnel. An information system should support these processes by enabling a seamless information flow between different participants and different locations. Because of rapidly changing and newly arising requirements, an appropriate information system should be capable of incrementally evolving according to the users' needs.[1] Yet current healthcare information systems stay far behind the expectations and rarely fulfil these requirements.[2–4]

Failures of IT projects in healthcare are not uncommon.[2–6] The deeper reasons for these failures are still subject of interdisciplinary research,[5,6] but many core factors are already well known. Inaccurate understanding of the end-users' needs is a symptom which is most frequently mentioned.[5,7,8] This can be led back to various reasons, such as insufficient communication,[8,9] and inability of users to express their requirements and lack of common ground.[3,5,10] Projects also often fail because of missing or unsuitable management models for software engineering.[7,8] The widespread waterfall model, for example, is insufficient for health IT projects because important assumptions for the applicability of this model are not fulfilled in the healthcare domain [11] (e.g., systems are required to evolve over time, and functional requirements are subject to change). Iterative software engineering[7–9] is more suitable for IT projects in the healthcare domain because it is aimed at incrementally developing and improving software products step by step. Iterative software engineering is also particularly suitable to be combined with a participatory approach, as postulated by many authors,[12] because it helps to encourage user feedback.[7] This effect is intensified if prototypes can be generated quickly without investing too much encoding effort. Thus, the approach is ideally supported by using a CASE tool for rapid application development (RAD).[13]

In healthcare environments, however, rapidly developing isolated applications is not enough because new applications are required to be integrated with the overall healthcare information system. This arti-

Affiliations of the authors: Institute of Medical Informatics, Philipps-University, Marburg, Germany (RL, TE, KAK); GWI Research Council GmbH, Vienna, Austria (HS).

Correspondence and reprints: Richard Lenz, DrIng, Institute of Medical Informatids, Philipps-University, Marburg, Bunsenstrasse 3, D-35037, Marburg, Germany; e-mail: <lenzr@mailer.uni-marburg.de>

Received for publication: 9/17/01; accepted for publication: 6/6/02.

cle describes such a CASE tool (*generator tool* or *forms generator*), which is an integral part of a commercially available hospital information system and which is particularly designed for generating workflow-enabled clinical applications that are integrated with the overall system. This generator tool is based on a document oriented user interaction paradigm. Users interact with the system by filling paper-like electronic documents, which serve as containers for information distribution (e.g., a discharge report) or a "laboratory order." The generator tool allows to rapidly develop new document types (templates). We refer to such a template as an "electronic form." In other words, an electronic document is an instance of an electronic form. Workflow is supported by establishing a flow of electronic documents between different parties involved in the health care process (e.g., physicians in different departments, nurses, medical-technical assistants, secretaries). The concept and its implementation are explained in detail in the main section of the article. In particular, we describe the underlying generic database schema.

This approach is being used in our university hospital, where the tool has been continuously improved during the past two years. Although the current implementation has certain limitations, the approach has proven practicable, and we consider the design ideas based on our experiences helpful for system architects and for implementers.

Background

Supporting clinical processes with information technology requires workflow specification (i.e., the identification of tasks, procedural steps, input and output information, people and departments involved, and the management of information flow according to this specification). Different approaches for process support in health information systems are currently under discussion. The most comprehensive approach for process support is offered by dedicated Workflow Management Systems (WfMS).[14,15] General purpose WfMSs try to provide a complete environment in which to define, execute, and monitor business processes. Existing applications are linked together into global processes that are controlled by an external workflow engine. The WfMS approach is conceptually mature and successful in many application areas. Well-structured business processes are increasingly supported by WfMSs. However, the approach does not yet play a significant role in health information systems.[16] One reason might be that existing clinical applications often come with built-in workflow functionality, which is not conform to standards for interoperating workflow engines as proposed by the Workflow Management Coalition (WfMC).[17] More important, however, is that integrating autonomous, independently developed applications in healthcare takes a high effort, no matter which integration approach is used. Independently developed applications tend to be technically and semantically incompatible.[18] Thus, getting up a WfMS in the extremely complex healthcare environment is risky (integration efforts are often underestimated), costly, and time-consuming.

Architectural flexibility in health information systems could be achieved by component-based systems with exchangeable plug and play components. Emerging component technologies, such as CORBA, DCOM or EJB, are often considered the key to integration of heterogeneous and autonomous components. Consistently combining domain-specific "best of breed" components into a comprehensive integrated whole seems to be within reach. This, however, is still not the case. Object middleware standards such as CORBA provide a framework for the development of distributed applications, enabling the programmer to concentrate on application logic instead of distribution issues such as remote access or distributed transaction management. Developing distributed applications on top of such a framework, however, still requires conformance of different components involved with a common application framework. The essence of component technology is to build well-structured systems out of independently understandable and reusable building blocks.[19] To make a component reusable by other components, a precise specification of syntax and semantics of the components' interfaces is required. If the interfaces of independently developed components do not match semantically, components will not cooperate—even if they are implemented in the same component technology. To achieve a truly integrated component-based system, component developers need to build their components on the basis of a common context. This common context, in particular, includes a functional decomposition of the application domain. Moreover, common ontological and terminological foundations of the application domain are required to achieve a common understanding of basic domain specific concepts. This problem is well known, and there is already a clear trend toward domain-specific middleware in healthcare.[20–23] Yet standardization efforts have not yet resulted in a complete application framework generally accepted and used for healthcare information systems. System evolution by

adding plug-and-play components in a "best-of-breed" strategy is still difficult.

We report on a generator-tool approach that supports a different strategy for incremental system evolution. The generator tool is an integral part of a holistic health information system, and it is used to rapidly develop new functionality for this system employing an iterative and participatory software engineering process. Instead of trying to link independently developed applications together, the generator tool provides a means for developing new system components, which are embedded into the overall system from the very beginning. As it is still not realistic to provide the complete spectrum of clinical functionality by the use of such a tool, conventional ways of connecting subsystems are also needed. In our hospital (and in other hospitals using software based on the generator approach), typical standard HL7 interfaces and an interface engine are used to connect separate subsystems. The number of separate subsystems tends to be limited, however. As an example, our laboratory system is connected via HL7, whereas the radiology information system has already been built with the generator tool.

Design Objectives

The objectives of the generator tool approach are to improve the quality of software components by bringing software development closer to the end-user. The tool is intended to ease the apposition of new functions into a running health information system and thereby to shorten development cycles and increase adaptation to the end-users needs. To achieve these goals, development of a new form should not be overloaded with routine technical programming tasks that are not directly related to the application logic (e.g., mapping of different data formats). With this approach, software development is no longer a two-stage process in which some programmer implements the requirements of the end-user. Instead, an additional distinction is needed between the system developer and the application developer:

- The **system developer (system architect)** is responsible for developing and improving the overall system architecture including the generator tool and its functionality. With respect to the implementation of the generator tool, the architect is not so much interested in the concrete requirements of some clinical department but merely in more general abstract requirements concerning the design of clinical applications. The system architect receives feedback primarily from application developers, not from end-users.
- The **application developer (application designer)** uses the generator tool for developing new applications employing a participatory and iterative software engineering process. He or she gets feedback from the end-user. The generator tool should help the developer to rapidly implement prototypes according to the end-users' requirements without investing too much encoding effort (e.g., the application designer does not need to worry about physical database schema design). This role is closely related to knowledge engineering. The application developer captures and formalizes domain knowledge. In the light of the fundamental works of Musen on Protégé, a central task of the application developer is extraction of a domain ontology.[24,25] Thus, the term *knowledge engineer* may be used from this perspective.
- The **end-user** (e.g., physician, nurse) is the person who works with the resulting applications (tool-generated applications). To optimize the adaptation of the software to the clinical work practice, the end-user is intensively involved in the software engineering process. Prototypes illustrating the design of forms and of information flow are available early, and they allow a continuous feed back from the end user to the application developer.

This article focuses on implementation aspects of such a generator tool. Thus, we primarily take the viewpoint of the system developer, who has been developing a tool suitable for rapidly implementing distributed clinical applications that support clinical processes including cross-departmental workflows. The generator tool should enable the application designer to concentrate on the application logic instead of encoding problems. Most importantly, however, the generator tool has to support the definition of applications that are integrated with the overall health information system via a common database. Thus, the main objectives for the system designer are as follows:

- Newly defined data items should be stored within the same central database, and newly defined applications should be accessible via a common framework without any *a posteriori* integration effort.
- Tool-generated applications should avoid redundant data entry. Therefore, the tool must enable the application designer to refer to existing data elements that are already stored in the central database to upload these data into newly designed applications.

- The tool should support developing distributed (interdepartmental or interinstitutional) applications such as order entry/result reporting. Therefore, the tool must provide means for workflow specification—i.e., means to specify how to deliver the right information (what), at the right time (when), to the right people (who).

Perspectives of a Document Based Generator Tool

The concrete approach in this article is based on a document-oriented user interaction paradigm, which substantially determines the different perspectives of end users, application developers and system developer. For this article we use the terms related to this paradigm as follows:

- *Electronic documents* serve as a structured unit of information and information flow.
- Each electronic document is an instance of an *electronic form*, which in turn is to be seen as type or template for corresponding electronic documents.
- A form contains different *fields*.
- An *entry* is an instance of a field that belongs to a concrete electronic document.
- *Reference lists* contain *references* pointing to electronic documents or to other reference lists. They are the point of ingress to electronic documents and an important instrument for gaining overview. A reference is represented by a short description of the contents of the associated document (e.g., discharge letter for patient John Doe from June 21st, 2001). Different types of reference lists can be distinguished as follows:

  1. **Patient-related lists** contain references to patient related documents. Examples are the patient history, in which all documents for a particular patient are collected, departmental views on a subset of the documents that belong to a particular patient, and reference lists for open requests for a particular patient.

  2. **Task-related reference lists** are work lists that are related to a specific task (e.g., discharge reports that are to be validated).

  3. **Meta-level reference lists** contain references to other reference lists. Examples are department overviews that contain references to patient specific lists, and user specific lists that contain references to different lists which are at the users disposal.

The generator tool is a means to rapidly develop new document-based applications. The generator tool and the tool-generated applications are embedded into an application framework, which is based on a common central database. The application framework maps both document data and necessary meta-data to generic database tables. These generic tables are one part of the overall database schema of the health information system, which also contains conventional database tables with explicitly modeled semantics. Administrative applications (e.g., patient data management with ADT functionality, financial accounting) operate on these conventional database tables. To illustrate the basic idea, the overall system architecture is shown in Figure 1.

The Perspective of the Application Developer

The development of tool-based applications must be seen in the broader context of an iterative and participatory software engineering process. The application designer closely cooperates with the end-users to capture specific workflow requirements and to adapt the application to the users' needs. This is done by elaborating the required information flow and mapping it to workflow enabled forms and reference lists. The generator tool supports the development of new forms and new reference lists.

Reference lists are also used to control access rights. The application developer assigns different Roles to different user groups. Each role is associated with access rights to certain reference lists. If a new form is to be developed, the application designer figures out who is going to use this form (which user groups are involved) and which reference lists are needed as ingress points for these users. If necessary, new reference lists are to be defined. For each new form, the application designer defines the associated fields and their graphical layout. The application designer may also define a default value or a computation rule for a field. Access restrictions can be defined for certain fields (e.g., only a physician is allowed to validate a discharge letter). To specify document flow, different states are to be defined for each form. An electronic document has exactly one state at a time. The application designer specifies in which reference lists a document appears in a certain state. For example, the reference list "new results for John Doe" presents an overview of all documents containing results for this patient (e.g., laboratory results, radiology results,

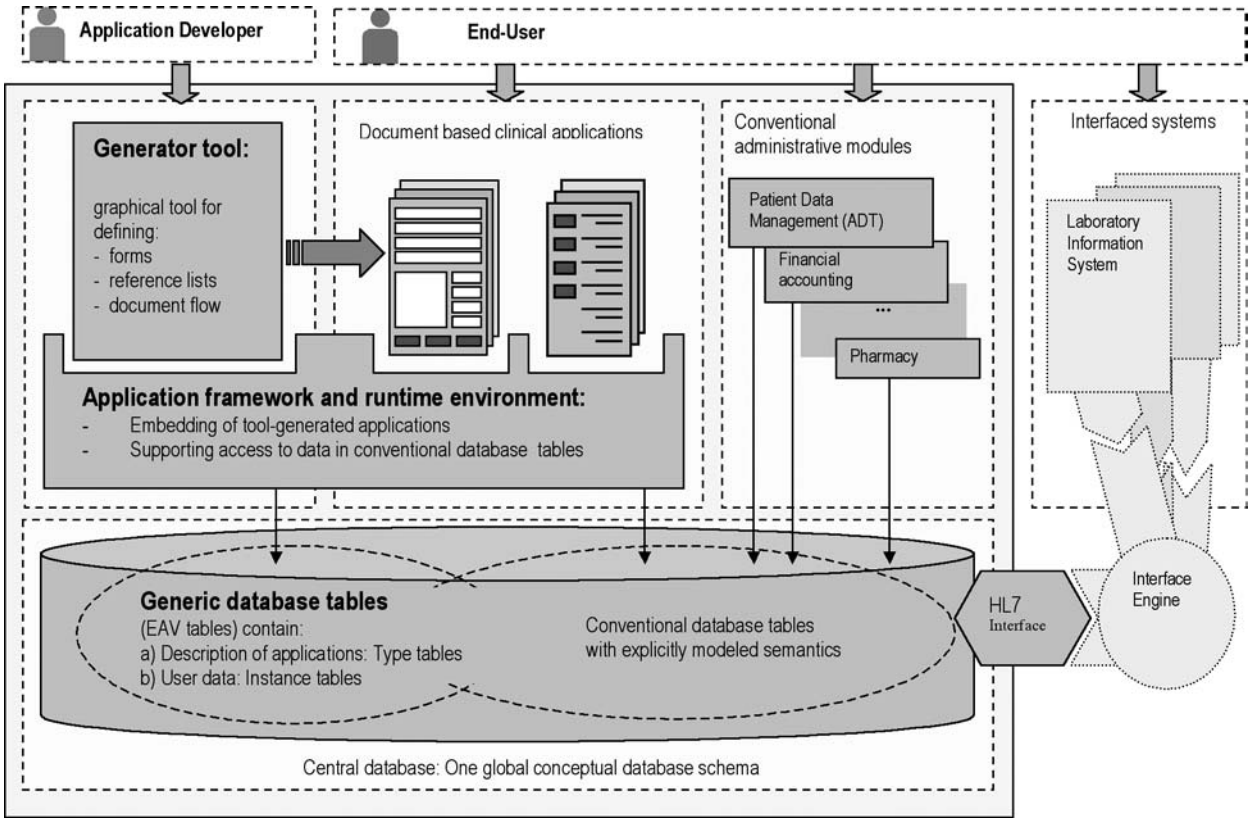

Figure 1 Overall system architecture. Conventional administrative applications as well as document-based clinical applications directly operate on a common central database. An application framework provides access to generic database tables as well as conventional (explicitly modeled) database tables. A generator tool is used to incrementally add new clinical applications to the system, which are based on a document-oriented user interaction paradigm.

reports from consultants) that have not yet been seen by the ordering physician. Moreover, the application designer defines the triggering events that change the document status. Each user interaction (e.g., updating a field) is an event that could be used. However, typically an explicitly defined button is used as a triggering event for a status change (e.g., a button "validate"). A trigger may also initiate an automatic upload of information from the central database. In terms of our initial characterization of workflow specification, document based applications as described cover all three workflow aspects:

- "What" is covered by the definition of forms and their contents
- "Who" is covered by assigning access rights to reference lists as well as individual fields and buttons
- "When" is covered by defining triggering events for controlling document flow

To enhance software reuse, it is also possible to include repeatable fields and sub forms within electronic forms. The application developer may, for example, define a sub form for entering a diagnosis and reuse this sub form in multiple "top-level forms." Workflow, however, is only specified for top-level forms.

The Perspective of the End User

Each user enters the system via his personal user folder, which provides access to the reference lists for which the user is authorized. By opening department- and patient-specific reference lists, the user navigates into a specific context. For example, Dr. A views the departmental patient history of Cardiology for patient John Doe. If the user creates a new document by instantiating a form, this current context information is used to automatically upload context specific information into the document (e.g., a newly instantiated discharge letter is

automatically filled with the patients address and other context determined information). Once a document is open, the user typically interacts with the system by filling this document almost like an ordinary paper form, and by explicitly invoking events (e.g. clicking on a dedicated button) to send this document to its destination. Electronic documents are typically filled step by step at different locations and by different users. A radiology report, for example, traverses different steps before it is completed and validated. Changes of validated documents are not permitted. If modifications are necessary, a new version of this document can be created while the old version is kept with an explicit invalidation remark.

Technical View on Electronic Forms

Technically, a form is an event-driven program that collects information from multiple sites (different locations involved in a distributed health care process) and stores it persistently in the central database. This program is always executed in a certain context: When a user opens a discharge letter for John Doe, the corresponding program is executed, and the current context information for John Doe and the document contents are uploaded into the local program variables. The program starts an event handler, which is simply a routine which registers events (such as user input) and performs predefined actions, accordingly. Each form has a closing event. If the event handler registers the closing event, the document contents are written back into the central database and the program terminates.

System Description

This section describes implementation aspects of the generator tool approach. In particular we describe the underlying generic database schema, which covers both the description of document-based applications as well as the data on which these applications operate.

Generic Database Schema for Document-based Applications

An extensible system, capable of dynamically handling newly defined forms and reference lists, requires the underlying database to be designed in a way that allows introducing new concepts without modification of the existing database schema. The common approach is an entity-attribute-value (EAV) database schema, as described by Nadkarni et al.[26] The basic idea of EAV is to use a single database table for arbitrary data items. The EAV table basically contains three columns—one column to specify the entity, one to specify the attribute, and one for the value of the specified attribute. Data items with newly defined semantics can be entered into the database without the need for a schema modification. Necessarily, semantic control on the database schema level is given up. Instead, semantic annotations are entered into explicitly introduced tables for such metadata.

In our case, a modified and extended EAV schema is required which also contains generic tables for the description of dynamically defined document based applications. We need both type level tables for description of tool based applications (e.g., forms and fields) and instance level tables for storing electronic documents and their contents. Type level tables are filled by the application developer via the generator tool. Instance level tables are filled at runtime by the end-user via the use of tool generated applications. The schema fragment shown in Figure 2 contains tables for forms, fields, documents, and entries. The tables *Form* and *Field* both describe the structural type of electronic documents (e.g., the form "discharge report" may contain fields for patient data, physician address, date, and comments). The table *Document* represents form instances. A document has particular entries (instances of fields). The table *Entry* is the equivalent of the classic EAV table that actually contains the user data to be stored. Instead of columns for entities and attributes, however, the table contains columns for documents and fields, respectively.

This construct allows association of entries with documents and fields, but it is too weak to identify semantically interrelated fields in different forms. For this purpose an additional layer of metadata tables for semantic annotation of fields has been introduced. Following Chen's E/R notation, we have named the corresponding tables *Entity*, *Relationship*, and *Attribute*. The comparable meta-data tables in Nadkarni´s EAV/CR schema are named *Classes, Class_Hierarchy,* and *Attributes*, following an object-oriented notation.[26]

The concept supports reuse of subforms in multiple top-level forms. Subforms can be recursively included in other forms by allowing fields of the type "subform-reference." If a form is modified, it is essential to keep the old version of the form in order to still be able to interpret older documents. Additional tables *Form_Version* and *Layout* are used to support form versioning. They describe which field is used in

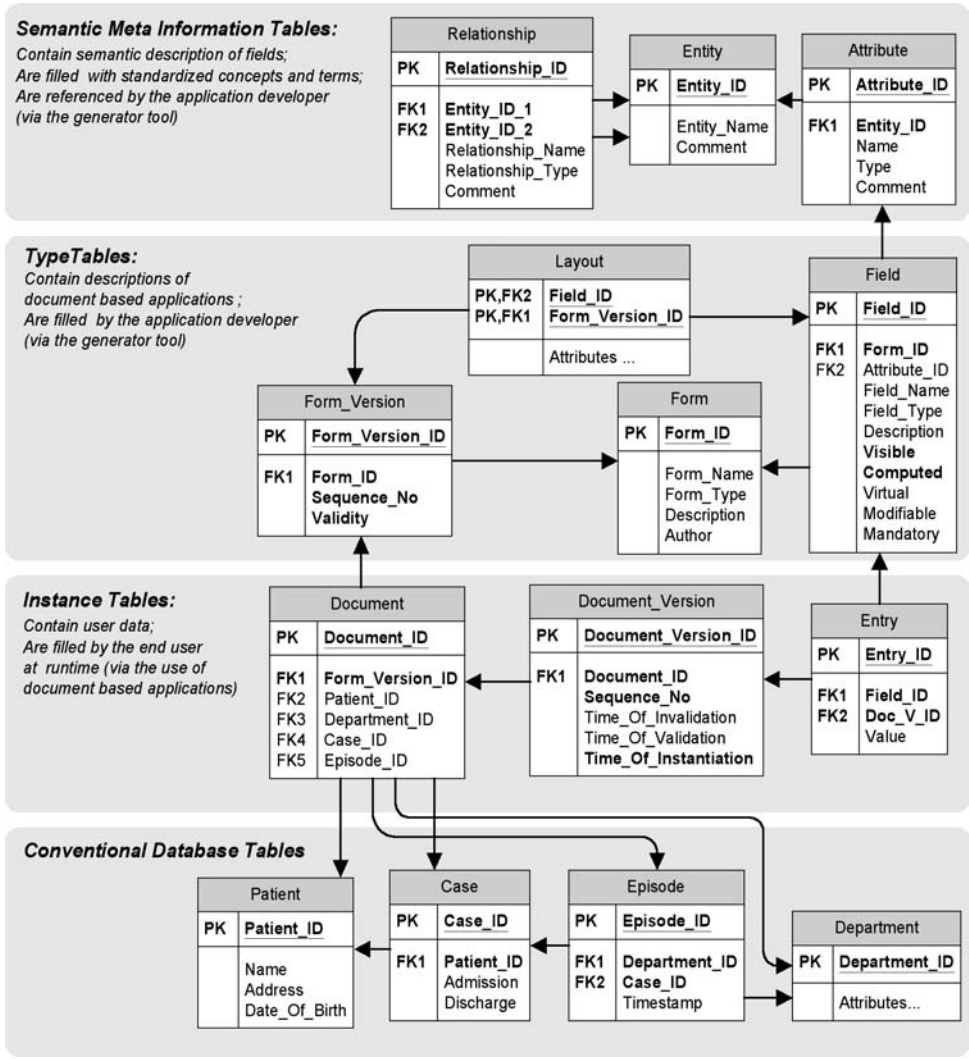

Figure 2 Fragment of an EAV schema for storing forms and electronic documents. Primary key (PK) and foreign key (FK) attributes are labeled. Mandatory attributes are indicated by bold font. The tables *Form, Form_Version, Layout* and *Field* are used to describe document types. The tables *Document, Document_Version* and *Entry* are used to store document instances. The table *Entry* is the actual EAV table, which is used to store all kinds of data items regardless of their semantics. Semantic annotations are stored in the tables *Entity, Attribute*, and *Relationship*. Electronic documents (table *Document*) are embedded into a clinical context by foreign keys to conventional database tables *Patient, Case*, Episode and Department.

which form version and with which layout (e.g., position, size, color). As a validated electronic document must not be modified, versioning is also needed on the instance level: An Entry is always associated with a document version (*table Document_Version*).

Generic database tables based on the EAV approach are used to be able to dynamically add new forms and fields to the system and thereby to extend the system's functionality. Advantages and disadvantages of the EAV-approach are well known,[26] and it makes sense to use conventional database tables with explicitly modeled attributes for major (static) parts of the schema, especially for patient administration data including the ADT-data. This conventional part of the database schema particularly contains database tables needed to describe the clinical context information in which electronic documents are to be embedded (e.g. an electronic document is generated in a certain department and belongs to a patient and a case). The context of a document is expressed by directly referencing these tables: The table *Document* actually contains foreign keys referencing conventional database tables such as *Patient*, *Case*, *Episode*, and *Department*.

Means to Support Data Integration

One of the main characteristics of the generator tool approach is the ability of the tool to enable the application designer to include references to existing data items into newly defined forms. These may either be data items stored in conventional database tables (e.g., ICD diagnoses) or even data items from the EAV table that belong to some other electronic document. This is an essential feature of the generator tool

that enables the application designer to develop truly integrated applications from the very beginning. Subsequently, we refer to a field which is derived from some other data item as a *computed field*. First, we concentrate on computed fields referring to data stored in conventional database tables. A possible solution would be to allow the application designer to specify SQL-statements that compute the desired information for an electronic document. Direct SQL-access to the conventional database tables, however, is not a good idea because the definition of a form would depend fully on the structure of these tables, and old forms might become unusable with a new software release. Therefore, to be prepared for schema evolution, it is better to encapsulate conventional database tables into objects, which are used by the application designer to refer to the associated information. If the definition of a conventional database table is modified in this case, it is necessary to update only the encapsulating object appropriately to adapt computations for all documents accessing this table. Another positive effect of this method is that the application developer is not confronted with the whole complexity of the database schema. Instead, the developer can choose from a predefined set of objects which make sense in the current context.

After considering computed fields derived from conventional database tables, we should look at the second case: A computed field may also be derived from another field in some other form. A typical case is that an entry from a completed and validated document is to be displayed in a newly created document. For example, findings from different examinations are to be uploaded in a discharge report. The generator tool offers a variety of options to upload contents from other documents: The application designer can either directly identify the document he refers to by computing its document ID (e.g., explicitly specifying an SQL query, which delivers the document ID), or chose from a set of predefined context dependent options (e.g., "last document of this case," "last laboratory report for this patient").

Different strategies of storing and updating entries for computed fields are to be distinguished. One option is not to store the entries redundantly in the EAV table (table *Entry*). We refer to such a field as *virtual field*, and a corresponding entry is called *virtual entry*. A virtual entry is automatically computed and uploaded into local program variables when the document is created or opened, and it is written back to the source database table when the document is closed. There is no further synchronization with the source database tables, unless the application developer explicitly specifies trigger actions for repeating the upload or for writing back to the database. As the contents of validated documents must not be changed, virtual entries automatically imply the need for versioning of conventional database tables. An update of a data item in its source database table must not lead to an update in old and validated documents referring to this data item. If the conventional database tables are not versioned, the application developer must ensure that virtual data entries are substituted by redundant entries in the *Entry* table as soon as the document is validated. An alternative with less redundancy is to delay explicit storage until the referenced data are modified.

If a computed entry is explicitly stored in the *Entry* table, it is up to the application developer to decide how to synchronize this entry with the source data from which it is derived. If the computation is used only to set a default value for this field, further updates or synchronization with the source database tables are not necessary (we refer to this case as an *unsynchronized computed field*). If the field, however, is semantically redundant with the source database table, the application designer must take steps to prevent diverging instances of redundant data. The generator tool has to support this goal by providing means to specify an appropriate replication protocol. In its current version, the generator tool only provides asynchronous mechanisms for update propagation between redundant entries. For computed fields referring to conventional database tables, the application programmer typically chooses a primary copy approach, in which the data item in the conventional database tables is the primary copy (master) and all redundant entries in the EAV table are the secondary copies (slaves). This approach avoids diverging instances of the same data since updates are always performed on the master copy. Bernstein et al.[27] provide a more detailed discussion of the underlying synchronization issues.

As shown in Figure 2, the table *Field* may contain several flags to describe the storage and update characteristics of a particular field:

- *Field_Computed*: is true if corresponding entries are computed from conventional database tables.
- *Field_Virtual*: is true if corresponding entries are computed but not redundantly stored in the table Entry.
- *Field_Modifiable*: is true if the user is allowed to explicitly update the field.

- *Field_Synchronized*: is true if the field is computed and modifiable and updates on the field are to be synchronized with the original conventional database tables.

Another important attribute to characterize a field is *Field_Visible*, which is true if the value of the corresponding data entry is actually displayed on the electronic document. Invisible fields are typically defined for additional document-related status information that is not interesting for the user of the document but that is used by the application designer for internal document control.

The attributes described in this section and in Figure 2 clarify the concept, but they are not sufficient for a concrete implementation of a generator tool: Additional tables and attributes are required; e.g., for specifying whether a field is updated continuously (immediate asynchronous propagation) or not. Furthermore, to describe delayed updates, specification and storage of triggering events are needed, but not covered by the schema fragment in this article.

Handling Dynamic Aspects of Electronic Documents

To support document flow, the generator tool must provide means to specify when a document appears in which reference list. For each form, a status variable has to be defined and initialized, trigger events for state changes have to be specified, and reference list assignments have to be made according to the current status. In the current implementation of the generator tool, all this is done explicitly by the application developer:

- The developer explicitly defines an invisible field interpreted as the document status.
- The developer specifies trigger events that initiate modifications of the status field.
- For each relevant reference list, the developer defines how references for this form and this list are represented. The developer also defines a condition indicating when a document of this type appears in this reference list. A typical condition would check whether the status field has a certain value (e.g., $status == "validated"); arbitrary conditions are possible, however.

As a consequence of this method, document definition and workflow definition are closely intertwined and implementation of document flow is cumbersome. The application designer must explicitly interpret the individually defined status variable and define trigger actions for state changes and reference list assignment. Once processes are implemented and hidden in the definition of forms, they are difficult to understand and to modify.

Current developments are directed towards a more comfortable way of specifying document flow. For this purpose, specification of document flow and specification of form contents are separated. Workflow specification via the generator tool can be eased by the use of graphical elements (e.g., by using simple directed graphs that are annotated with attributes for workflow definition). A graph contains nodes and directed edges. A directed edge connects a source node and a destination node. Nodes represent workflow states and are associated with reference list types. A directed edge indicates a possible state transition. "Empty forms" that already include triggering events and reference list assignments can then be automatically generated from such a graphical description of workflow. Within these empty forms the application designer can already refer to workflow states in order to use them for document internal control (e.g., depending on the state, the application designer might want to decide whether a certain field is modifiable or not). Besides ease of programming and advanced maintainability, another advantage of this approach is that the workflow specification is automatically documented.

The relational database schema described so far covers the contents of forms and documents only (the "what" in the workflow specification). Additional tables are required to store reference lists, references, and document states. These extensions are described in Figure 3. The characteristics of a reference list are described by a reference list type (table *Reference_List_Type*). Concrete reference lists are instances of a reference list type and are stored in the table *Reference_List*. Like electronic documents, reference lists can be related to a concrete clinical context. A reference list type "patient history," for example, has concrete instances for each patient.

With the database schema in Figure 3 we present the current developments toward a more stringent interpretation of the document-based approach. Definition of document flow is supported explicitly, and it is separated from the definition of form contents. Each form (table *Form_Version*) is assigned to a number of workflow states (table *Doc_State*). The table *Document* receives an additional foreign key on the table *Doc_State*, indicating the current state of the

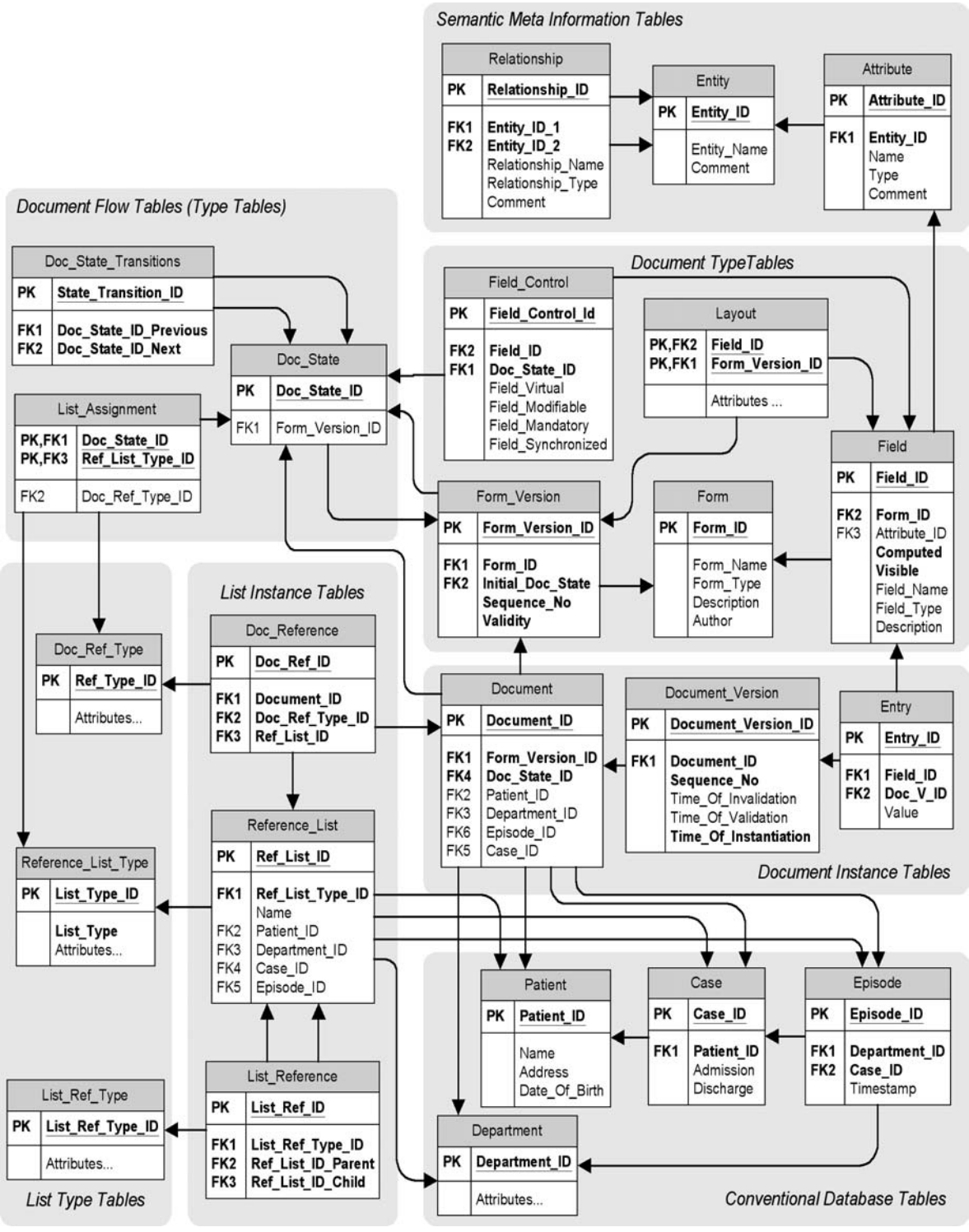

Figure 3 Extended EAV schema for storing workflow-enabled forms and electronic documents. Tables for storing reference list types and reference list instances have been added. The context of a reference list instance (Table *Reference_List*) is expressed by foreign keys to conventional database tables *Patient, Case, Episode*, and *Department*. Document states (Table *Doc_State*) are used as core element to describe the dynamic behavior of documents, i.e. state transitions (Table *Doc_State_Transitions*), dynamic reference list assignment (Table *List_Assignment*), and dynamic changes of field attributes (Table *Field_Control*).

document. The document flow for a certain form is described by defining the potential successor states for each document state (table *Doc_State_Transitions*). When a document is used at runtime, trigger buttons are displayed for each successor of the current state of this document. For each document state, it is necessary to define in which reference lists a corresponding document is to be displayed (table *List_Assignment*). The table *List_Assignment* refers to a document state and a reference list type (e.g., patient history); the concrete reference list instance for a concrete document is determined by the context of that particular document (e.g., if the document belongs to patient John Doe, it will appear in the patient history for John Doe). Task-related work list types typically have only one instance. Reference types are used to specify how a document is referenced within a particular reference list (table *Doc_Ref_Type*); for example, a validated discharge report may be referenced by displaying the document type, the patients name and the date of validation. A concrete reference is an instance of this type (e.g., "discharge letter for patient John Doe from June 21, 2001"). A list assignment contains an additional foreign key to the Table *Doc_Ref_Type* to determine which reference type is to be used for a document in a certain state and in a certain reference list. The table *Doc_Reference* is used for concrete document references (e.g., document A appears in reference list B using the document reference type C). Meta-level reference lists contain list references instead of document references. The type of such a list reference is described in the table *List_Ref_Type*. Concrete list references are then described in the table *List_Reference* (e.g., list A appears in list B using reference type C).

Now that document states are explicitly stored in the database, they can be used to control dynamic field attributes: For example, a field may be modifiable until the document is validated and not modifiable from then on. The table *Field_Control* is used to store the definition of this state dependent dynamic behavior.

Practical Aspects of Implementation

The relational database tables presented in this article are intended to explain the concept of a document-based generator tool. As already mentioned, they do not represent a complete database schema, which is needed for a concrete implementation. Additional tables for event handling are needed, additional attributes for layout and field positioning and the like are needed, and the conventional database tables are indicated only by rudimentary table fragments. Moreover, in order to achieve an acceptable performance further considerations are necessary. For example, interpreting forms solely from their description in relational database tables will result in unacceptable response times. Therefore, it is necessary to provide measures to increase performance, e.g. redundant storage of pre-generated forms as blobs.

Status Report

From the vendor's perspective, the generator tool is the fundamental strategy by which all new clinical applications are developed. The tool can be used in different scenarios. The application developer can either be a member of the hospital's IT staff or one of the vendor's specialists. A further perspective might be to hand the tool to the end-user to let him or her design his or her own forms; a modified (simplified) version of the tool is being discussed for this purpose. While tool-generated applications are in use in more than 200 hospitals in Germany, only a few hospitals are developing their own tool-based applications. Instead, the vendor company employs both a (smaller) group of system developers and a (larger) group of application designers.

In our university hospital, the tool has been in use since December 1999. Major components of the hospital information system, such as clinical applications for wards and ambulatory care settings as well as ancillary systems (radiology, pathology, endoscopy and others), are based on the generator tool approach. Several of the generator-based distributed applications have been implemented by our hospital IT-staff.[28] Generator-based applications are currently being used at all wards throughout the hospital (approximately 180 clinical workstations at 90 wards). Additional generator-based applications are used at approximately 100 workstations (radiology, 40; nuclear medicine, 20; pathology, 25; dermatohistology, 5; endoscopy, 5; sonography, 4). Applications for several ambulatory care settings are being introduced. All ICD-diagnoses are captured via synchronized electronic forms and are directly written into conventional database tables. Documentation of procedures is also increasingly covered by electronic forms. Approximately 11,500 examination reports per month are produced by means of electronic documents (e.g., in August 2001: radiology, 8,162; pathology, 1,793; nuclear medicine, 993; dermatohistology, 230; endoscopy, 322). Currently, 10 of 30

departments use generator-based discharge reports (approximately 1,800 reports per month). Order entry for radiology has been implemented via workflow enabled forms, which include electronic scheduling. The introduction time was about 4 months for all wards (90) of the hospital. The central EAV table currently (Dec 2001) contains more than 42 million entries and is growing at an increasing rate of currently 2.2 million entries per month. From our experiences, neither the tool-generated code nor the EAV storage technique affected the system performance negatively. There were negative performance effects, but these resulted from the holistic system with dependencies on administrative modules (and long running administrative statistics programs); they were not caused by the generator approach.

The current implementation of the generator tool has limitations. As already mentioned, specification of document flow is not yet represented in the database. Instead, the application designer is forced to use invisible fields as status variables. This approach is unsatisfying because definition of document contents and workflow are intertwined and there is no possibility to easily obtain any workflow-related information from an existing generator-based application. Because definition and interpretation of status variables are up to the application developer, dynamic behavior of fields is also hidden in the program code. The generator tool is currently being re-implemented to separate workflow specification from the specification of document contents. This re-implementation also considers more general workflow issues, such as dependencies among different forms. Another goal of the re-implementation is to enhance reuse of software-components (e.g., subforms). To reach these goals, the application developer should be able to refer to standardized sub form interfaces within the workflow specification (e.g., an electronic document should allow external access to its states in order to enable a workflow engine to modify these states).

The second major drawback of the current implementation is that semantic annotation of fields is not sufficiently elaborated yet; semantic meta-information tables are missing. Usually electronic documents are accessed as a whole via reference lists (e.g., patient history, ward specific view, departmental view), which is sufficient to support routine clinical work practice. Thus, semantic annotation of document *contents* has not been absolutely essential for the success of the generator. Reasons for introducing semantic annotations are well known, however,[29,30] and the product is actually being developed into this direction. Although the table *Field* may have an attribute for an informal description of a field, a more stringent method is needed to be able to identify identical or related contents in different forms and for expressing semantic equivalence. The method of choice is an additional layer of metadata tables (see Figure 2) in the database schema which is used to describe the semantics of forms and fields.[26] These tables contain descriptions of entities (medical concepts) and relationships between these entities (see Figure 2). The semantics of a newly defined field can be expressed by relating this field to the associated medical concept.

The next step should be to use the contents of the metadata tables for consistent semantic tagging of XML documents, which can be generated if data are to be extracted and transferred to other systems. This step, however, requires semantic annotation of all fields. Moreover, if such an XML document is to be interpreted by some external system, it would be advantageous to use standardized XML tags. Ideally, the metadata tables are to be filled with more or less standardized medical concepts (standard ontologies), so that the metadata tables finally provide a means for referencing a medical entities dictionary or repository.[29,30] Both semantic annotation of electronic documents as well as explicit workflow support are currently under development.

Discussion

We have presented a generator tool for rapid prototyping of document oriented and workflow-enabled applications. Both the tool and the tool-generated applications are integral parts of a health information system. The approach allows dynamic evolution of the information system by incrementally adding new applications to the running system. The document-based generator tool approach has several advantages and drawbacks. Advantages of the approach can be briefly summarized as follows:

- Clinical documentation is supported in an intuitive way by emulation of paper-like forms.
- These forms are routinely used by different parties involved in the care process, resulting in an effective support of distributed clinical processes (e.g., radiology order entry and result reporting from the wards could be realized in a surprisingly short time).
- EAV storage of forms data results in flexibility and extensibility without the need for schema modifications. New attributes, data types, and concepts can

be included dynamically into a running system. Moreover, EAV tables are space-efficient since unassigned fields do not allocate any memory at all.[26]

- The document- and reference list-based approach in combination with EAV has the advantage that new task related work lists can be dynamically included into a running system, which enables dynamic extensibility of functionality.
- Database integration of tool based applications avoids additional efforts for interfacing compared with independently designed applications.
- The generator tool approach speeds up application development because the application developer is not required to encode forms in a programming language. Templates (e.g., a predefined discharge report) can be reused and adapted to specific departmental needs.
- The generator tool used in our hospital has shown that the approach is workable.

Drawbacks of the approach are:

- Typical disadvantages resulting from the EAV technique also affect the document oriented extended EAV approach. For example, database mechanisms for integrity control cannot be used in the usual way. A significant amount of program code has to be generated by the generator tool to maintain integrity, which is normally ensured by the database system.
- Querying data from documents stored in EAV tables is more complicated than querying conventional database tables.
- Specification and synchronization of computed fields and redundantly stored data are essentially up to the application developer. He decides when to upload computed fields from conventional database tables into electronic documents. Thus, the application developer must carefully consider what he or she is doing in order to avoid undesired effects. Writing computed entries back into conventional databases tables, for example, bares the risk of potential inconsistencies because the conventional tables might have been updated since they were uploaded into the document, which would result in a lost update.
- Using the generator tool for rapid development of clinical applications involves the risk of inhomogeneous forms. Style guides are needed to avoid this. Moreover, to avoid unsynchronized semantically redundant forms, the tool should only be used in a controlled manner (i.e., different application developers working on semantically related forms should cooperate).

Future implementations of the approach might be more flexible by using XML to store electronic documents and appropriate DTDs (document-type definitions) to describe the contents of a document. This, however, is only a question of how to implement the generator tool and which format is used to make the data persistent. The idea of developing distributed applications by defining work lists and workflow-enabled documents remains unaffected by this decision. For the long-term archiving of electronic documents, additional mechanisms are needed to achieve authenticity and security in terms of availability, confidentiality, and integrity. An electronic signature and means for redundant storage in a software-independent format (e.g., pdf) are currently being introduced in our implementation.

The generator tool described is essentially a tool for computer-aided software engineering. Software engineering and knowledge engineering are inherently related. A common goal is to make domain models and ontologies more explicit and thus improve the maintainability and flexibility of computer programs.[25] Musen's work on Protégé has been fundamental under this scientific perspective: Protégé is a set of tools and a methodology for building knowledge-based systems that support the clear separation of different system-building tasks such as explicit modeling of domain ontologies and entry of content knowledge.[24,25] Although the approach in this article can also be seen as a contribution toward more tailorable and maintainable systems, the domain knowledge flowing into generator-based applications is not clearly separated yet. The semantic annotation of document contents and the representation of a domain ontology are only rudimentarily supported, and information flow is still hidden in the declaration of status variables. The current developments described in the status report are indeed aimed at better considering these knowledge engineering aspects.

Component technology is gaining importance as standards for technical and semantic interoperability are evolving and increasingly adopted by healthcare IT vendors. Some examples have shown that efforts in integrating heterogeneous system components can be successful if one can manage to map appropriately the constituents of the system to real world

processes.[31] However, healthcare information systems are still far away from plug-and-play reuse of autonomously developed generic components. Bridging incompatibilities between coarse grained and functionally redundant subsystems is still state of the art.

Projects such as RICHE and its successors, from NUCLEUS to SynEx have aimed at the development of domain specific open systems architectures.[21–23] They have significantly promoted standardization efforts, but have not yet led to a commonly accepted framework and widely available commercial products. The concept of Act Management developed in the RICHE project[21] is an interesting approach to handling patient-related workflow issues in health information systems. Actually, the status of a medical action or act is related to the status of electronic documents, which are used to support medical acts. Our current efforts to separate workflow definition from document definition are going into the direction of intelligent act management.

Setting up a health information system today requires a strategy for system evolution that is aimed at the convergence and integration of the systems components. The approach presented in this paper is intended to support the development of new and integrated components, and it is clearly based on a database centric system architecture, which is its strength and limitation at the same time. Implementing a comprehensive hospital information system with a single central database is not a realistic scenario. For various reasons, heterogeneous autonomous subsystems still have to be integrated into the overall system.[32] Currently these subsystems are to be integrated in the conventional way by using standard interfaces such as HL7 or DICOM and an interface engine; imported data are typically converted into an appropriate format and stored in the central database. From there, these data may be uploaded into electronic documents.

Some important issues, such as data retrieval, synchronization and versioning of conventional database tables, are still somewhat difficult to handle within the current implementation of the generator tool. However, as plug-and-play components and an associated component market are not yet available, this approach appears to be a workable step towards an incrementally evolving and yet truly integrated healthcare information system.

References ■

1. Kuhn KA, Lenz R, Blaser R. Building a hospital information system: Design considerations based on the results from a Europe-wide vendor selection process. Proc AMIA Symp. 1999; 834–8.
2. Dorenfest S. The decade of the '90s. Poor use of IT investment contributes to the growing healthcare crisis. Healthc Inform. 2000;17(8):64–7.
3. Kuhn KA, Giuse DA. From hospital information systems to health information systems: Problems, challenges, perspectives. Methods Inf Med. 2001; 40: 275–87.
4. Institute of Medicine. Crossing the Quality Chasm: A New Health System for the 21st Century. Washington, DC, National Academy Press, 2001.
5. Sauer C. Deciding the future for IS failures: Not the choice you might think. In Currie W, Galliers R (eds). Rethinking Management Information Systems. Oxford University Press, 1999, pp 279–309.
6. Anderson JG, Aydin CE. Evaluating the impact of health care information systems. Int J Technol Assess Health Care 1997; 13(2):380–93.
7. Kruchten P. The Rational Unified Process—An Introduction. 2nd ed. Boston, Addison Wesley, 2000.
8. Verstegen G. Projektmanagement mit dem Rational Unified Process. Heidelberg, Springer, 2000 [in German].
9. Beck K. Extreme Programming—Das Manifest. München, Addison Wesley, 2000 [in German].
10. Coiera E. When conversation is better than computation. J Am Med Inform Assoc. 2000;7(3):277–86.
11. Brender J. Methodology for constructive assessment of IT-based systems in an organisational. Int J Med Inf. 1999; 56(1-3):67–86.
12. Timpka T, Sjoberg C, Hallberg N, et al. Participatory design of computer-supported organizational learning in health care: methods and experiences. Proc Annu Symp Comput Appl Med Care. 1995;800–4.
13. University of California, Davis. Rapid Application Development. Available at URL: <http://sysdev.ucdavis.edu/WEBADM/ document/rad_toc.htm>. Accessed December 20, 2001.
14. Georgakopoulous D, Hornick M, Sheth A. An overview of workflow management. Distrib Parallel Dat. 1995;3:119–53.
15. Alonso G, Mohan C. WFMS: the next generation of distributed processing tools. In: Jajodia S, Kerschberg L (eds): Advanced Transaction Models and Architectures. Boston, Kluwer Academic Publishers, 1997.
16. Dadam P, Reichert M, Kuhn K. Clinical workflows—The killer application for process-oriented information systems? Proceedings of the Fourth International Conference on Business Information Systems, Posen, April 2000, pp 36–59.
17. Members of the Workflow Management Coalition. WfMC Pub-lished Standards Documents. Available at URL: <http://www. wfmc.org/standards/docs.htm>. Accessed August 22, 2001.
18. Lenz R, Kuhn KA. Intranet meets hospital information systems—the solution to the integration problem? Methods Inf Med. 2001; 40: 99–105.
19. Szyperski C. Component software. Harlow, England, Addison Wesley, 1998.
20. Spahni S, Scherrer JR, Sauquet D, Sottile PA. Towards specialised middleware for healthcare information systems. Int J Med Inf. 1999;53:193–201.

21. Frandji B, Schot J, Joubert M, et al. The RICHE Reference Architecture. Med Inform (Lond). 1994;19(1):1–11.
22. Kilsdonk AC, Frandji B, van der Werff A. The NUCLEUS integrated electronic patient dossier breakthrough and concepts of an open solution. Int J Biomed Comput. 1996;42(1–2):79–89.
23. Xu Y, D'Alessio L, Jaulent MC, et al. Integrating medical applications in an open architecture through generic and reusable components. Medinfo. 2001;10(Pt 1):63–7.
24. Musen MA. Domain ontologies in software engineering: use of Protege with the EON architecture. Methods Inf Med. 1998; 37(4–5):540–50.
25. Musen MA. Medical informatics: Searching for underlying components. Methods Inf Med. 2002;41(1):12–9.
26. Nadkarni PM, Marenco L, Chen R, et al. Organization of heterogeneous scientific data using the EAV/CR representation. J Am Med Inform Assoc. 1999;6(6):478–93.
27. Bernstein PA, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. Reading, MA, Addison-Wesley, 1987.
28. Lenz R, Elstner T, Blaser R, Kuhn KA. Experiences with a holistic health information system. Proc AMIA Symp. 2001: 952.
29. Cimino JJ. From data to knowledge through concept-oriented terminologies: Experience with the Medical Entities Dictionary. J Am Med Inform Assoc. 2000;7(3):288–97.
30. Zeng Q, Cimino JJ. A knowledge-based, concept-oriented view generation system for clinical data. J Biomed Inform 2001; 34(2):112–28.
31. Geissbühler A, Lovis C, Lamb A, Spahni S. Experience with an XML/HTTP-based federative approach to develop a hospital-side clinical information system. In Patel V, Rogers R, Haux R (eds): Medinfo 2001. Proceedings of the 10th World Congress on Medical Informatics. Amsterdam, IOS Press, 2001: 735–9.
32. McDonald CJ. The barriers to electronic medical record systems and how to overcome them. J Am Med Inform Assoc. 1997;4(3):213–21.