U

UML

▶ Unified Modeling Language

Uncertain Databases

▶ Probabilistic Databases

Uncertain Information

▶ Incomplete Information

Uncertainty in Events

Segev Wasserkrug
IBM Research, Haifa, Israel

Synonyms

Event uncertainty

Definition

Uncertainty in events is uncertainty regarding either the occurrence of an event, or uncertainty regarding the data values associated with an event. This uncertainty is a result of a gap between the actual occurrences of events in the real world, and the availability of knowledge regarding the events.

Historical Background

The first event-based systems were active databases, in which automatic actions were carried out as a result of database queries. This was done using the *ECA* (Event-Condition-Action) paradigm. However, in many such database applications, the events of interest were not the results of single queries, (e.g., insertion or deletion of data), but rather could be deterministically inferred from several such queries. To facilitate such inferences, event inference languages were defined. Initially such languages were specific to active databases (e.g., SNOOP and ODE). However, more general languages were developed suitable for implementing such deterministic inferences in any event-based system. These general languages resulted from the need to enable event driven behavior in a wide variety of application domains and became part of a wider area known as Complex Event Processing (CEP). As event driven applications became more complex, it became necessary to handle uncertainty regarding the occurrence and inference of events.

Foundations

For a system to implement event driven behavior, the system must be able to recognize all events of interest. However, in many cases, there is a gap between the actual occurrences of events to which the system must respond and the data generated by monitoring tools regarding these events. This gap results in uncertainty.

To understand this, consider a thermometer that generates an event whenever the temperature rises above 37.5°C. The thermometer is known to be accurate to within ±0.2°C. Therefore, when the temperature measured by the thermometer is 37.6°C, there is some uncertainty regarding whether the event has actually occurred.

Another gap between the actual occurrence of events and the information available to the system is caused by the following: The information regarding the occurrence of some events (termed *explicit events*) is signaled by event sources (e.g., monitoring tools such as the thermometer described above), while for other events, explicit notification is never sent (*non-explicit events*). An example of a non-explicit event is insider trading. Although insider trading

either does or does not take place, no explicit signal regarding such an event is generated.

For an event-based system to respond to non-explicit events, in many cases the occurrence of these events must be inferred based on the occurrence of other events. (The events based on such inference are termed *inferred events*.) To facilitate such inferences, several event composition languages have been defined that make it possible to infer non-explicit events based on a set of complex temporal predicates. In many cases, however, such inference cannot be carried out with certainty. To see an example, consider again the case of insider trading. While a single large sale of stock may not be indicative of anything suspicious, such a sale, together with a sharp change in stock prices of the company due to an announcement in the press may infer insider trading. However, such insider trading cannot always be said to have occurred whenever a combination of a sale event and a stock decline event occur.

Even when the inference rules may be deterministically stated, uncertainty associated with explicit events may also propagate to inferred events. To see this propagation of uncertainty, consider a rule (defined in an event composition language) that states that event e_3 must be inferred whenever an event of type e_2 occurs after an event of type e_1. Moreover, assume that it is known that both an event of type e_1 and an event of type e_2 have occurred, but there is uncertainty about the exact time of their occurrence because the occurrence of e_1 is known to be between time 2 and time 5, and the occurrence of e_2 is known to be between time 3 and time 7. Note that even though the rule is deterministic, the uncertainty regarding the occurrence times of e_1 and e_2, and the fact that the occurrence of e_3 must be inferred based upon these occurrence times, results in uncertainty with regards to the occurrence of the event e_3.

Dimensions of Event Uncertainty

It is useful to classify the uncertainty according to two orthogonal dimensions: *element uncertainty* and *origin uncertainty*.

Element uncertainty refers to the fact that event-related uncertainty may involve one of two elements:

1. *Uncertainty regarding event occurrence*: Such uncertainty is associated with the fact that although the actual event occurrence is atomic, (i.e., the event either did or did not occur) the event-based system does not know whether or not this event has, in fact, occurred. An example of this is insider trading. At any point in time, insider trading either was or was not carried out by some customer. However, an event-driven system can probably never be certain whether insider trading actually took place.
2. *Uncertainty regarding event attributes*: Even in cases in which the event is known to have occurred, there may be uncertainty associated with its attributes. For example, while it may be known that an event has occurred at some point in time, its exact time of occurrence may not be precisely known.

Origin uncertainty pertains to the two types of events (explicit and inferred) that exist in an event-based system. Due to these two types of events, there are two possible origins for uncertainty:

1. *Uncertainty originating at the event source*: When an event originates at an event source, there may be uncertainty associated either with the event occurrence itself or the event's attributes, due to a feature of the event source. An example is the limited precision thermometer described previously, where uncertainty regarding an event occurrence (i.e., the temperature being above 37°C) is caused by the limited measuring accuracy of a thermometer (i.e., the reading being accurate only to within ± 0.2°C).
2. *Uncertainty resulting from event inference*: Due to some events being inferred based on other events, uncertainty can propagate to the inferred events. This is demonstrated by the rule previously described, which is used to infer events of type e_3 based on events of types e_1 and e_2. In the aforementioned case, uncertainty regarding the occurrence of event e_3 resulted from uncertainty regarding the time that events e_1 and e_2 occurred.

Based on the above two dimensions, it is possible to define four types of event uncertainty. These are the following:

1. Uncertainty regarding event occurrence originating at an event source;
2. Uncertainty regarding event occurrence resulting from inference;

3. Uncertainty regarding event attributes originating at an event source; and
4. Uncertainty regarding event attributes resulting from event inference.

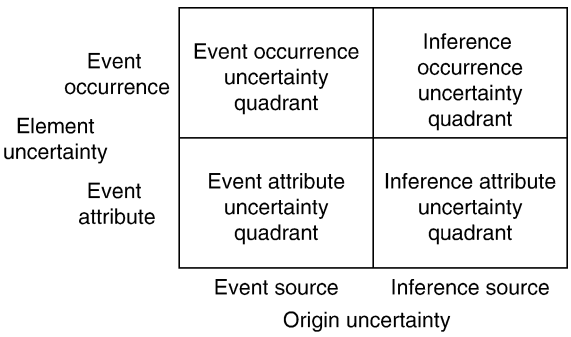
These uncertainty types are depicted as quadrants in Fig. 1.

Causes of Event Uncertainty

There are many possible causes for event uncertainty. In addition, the uncertainty causes for explicit events are different from the uncertainty causes of inferred events.

These are the sources of uncertainty for explicit events:

1. *An unreliable source*: An event source may malfunction, indicating that an event has occurred even if it has not. Similarly, the event source may fail to signal the occurrence of an event which has, in fact, occurred. A source may also transmit erroneous information regarding one (or more) of the event's attributes.
2. *An imprecise event source*: An event source may operate correctly, but still fail to signal the occurrence of events due to limited precision (or may signal events that did not occur). Such a source may also be the cause of imprecision regarding an event's attributes.
3. *Problematic communication medium between the event source and the event-based system*: Even if the event source has full precision and operates correctly 100% of the time, the communication medium between the source and the event-based system may drop indications of an event's occurrence, generate indications of events that did not occur, or alter information regarding the event's attributes.
4. *Uncertainty due to estimates*: In some cases, the event itself (or its attributes) may be the result of a statistical estimate. For example, it may be beneficial to generate an event whenever a network Denial of Service (DoS) event occurs, where the occurrence of such a DoS event is generated based on some mathematical model. However, as the mathematical model may produce erroneous results, this event also has uncertainty associated with it.
5. *Clock synchronization in distributed systems*: This is a cause of uncertainty regarding the occurrence time of events in a distributed system. This is due to the fact that in distributed systems, the clocks of various nodes are usually only guaranteed to be synchronized to within some interval of a global system clock. Therefore, there is uncertainty regarding the occurrence time of events as measured according to this global system clock.



Uncertainty in Events. Figure 1. Event uncertainty types.

These are possible causes of uncertainty of inferred events:

1. *Propagation of uncertainty*: There are cases in which an inferred event can be deterministically inferred based on other events. Even in such cases, there may be uncertainty regarding the inferred event, resulting from uncertainty regarding the events based on which the inference is carried out.
2. *Uncertain inference*: There are cases in which the inference is inherently uncertain. One example of this is insider trading, where events denoting suspicious purchases and sales of stock coupled with rapid price changes only serve to indicate the possible occurrence of an insider trading event. In such cases, an insider trading event cannot be inferred with certainty based on such suspicious transactions. An additional prominent example is when the event driven application is required to predict the occurrence of future events.

Note that for both explicit and inferred events, uncertainty regarding a specific event may be caused by a combination of factors. For example, for a specific event, it is possible that both the event source and communication medium simultaneously corrupt the information sent regarding this event.

Handling Uncertainty in Events

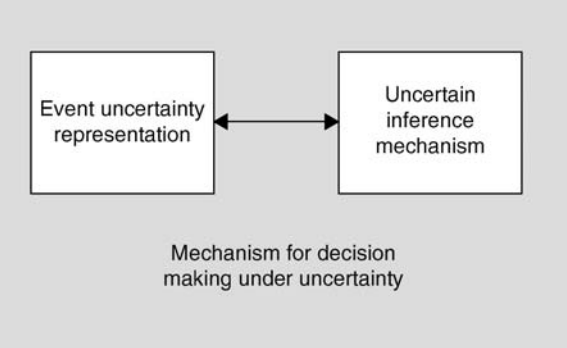
There is a spectrum of possibilities by which such event uncertainty may be handled. One end of this spectrum has methods that explicitly or implicitly ignore the presence of uncertainty. The other end is a complete and formal treatment of such uncertainty. Details about these two extremes are provided below. Obviously, solutions which lie between these two extremes are also possible.

The methods that explicitly or implicitly ignore such uncertainty usually rely on deterministic event composition languages to enable the event driven functionality required. In many cases, applications that ignore the uncertainty rely on a human being to make the final decision regarding whether or not an event of interest occurred. An example of such an application is credit card fraud detection. In many such systems, the system is expected to recognize suspicious patterns of credit card transaction events that may indicate fraud. Such indications are then used to alert a human operator, whose role it is to establish whether such a fraud indeed took place.

A full and formal treatment of such uncertainty is required in applications where automatic actions must be carried out as a result of events of interest. In such cases, the framework for dealing with event uncertainty must include the following components:

1. Mechanism for automatic decision making that can take uncertainty into account
2. Mechanism for enabling uncertain inference regarding events
3. Representation method of the uncertainty associated with each event

A depiction of such a framework appears in Fig. 2.



Uncertainty in Events. Figure 2. Elements of event uncertainty handling framework.

Example of Event Uncertainty Handling Framework

This section provides an example of an uncertainty handling framework that has the three components described in Fig. 2.

Underlying any framework for handling uncertainty in events is an uncertainty handling mechanism. Many such formalisms exist, including *Lower and Upper Probabilities*, *Dempster-Shafer Belief Functions*, and *Possibility Measures*. The framework discussed in this section is based on probability theory, which is the most well known framework for quantitative representation and reasoning about uncertainty.

In this framework, the uncertainty regarding an event is represented as follows: the event may have multiple sets of values associated with it. Each such set of values corresponds to possible attribute values of this event. The uncertainty is then quantified by probabilities for this set of values. For example, consider an event that either has not occurred, has occurred at time 5, or has occurred at time 10. This event has the set of values {*notOccurred*, 5, 10} associated with it. Furthermore, if the probability of the event not occurring is 0.5, the probability of the event having occurred at time 5 is 0.3 and the event having occurred at time 10 is 0.2. This is represented by {{*notOccurred*,0.5},{5,0.3},{10,0.2}}.

The framework enables uncertain inference by using a language that can specify uncertain rules together with an inference algorithm that enables the probabilities of interest to be calculated. The uncertain rules in the framework are of the form "If event e_1 and event e_2 occurred, then the probability of event e_3 occurring is 0.7." The inference framework ensures that the probabilities of the inferred events are taken into account in a manner consistent with probabilistic dependencies and independencies between the events.

The automatic decision making mechanism is based on utility theory.

Key Applications

Event driven functionality is required in almost all application domains. Therefore, the use of event-based systems is widespread. Two prominent examples of applications in which a complete treatment of

uncertainty is required are security applications and sensor network applications.

In security applications, the response time available to respond to threats requires that automatic actions be carried out. An example is network security applications where it is important to respond quickly to Denial of Service (DoS) attacks. There is a need to provide a quick response (e.g., in milliseconds) to the threat, since waiting too long to respond may mean that the network may already be too saturated for a response to be of use. On the other hand, wrongly determining that a DoS attack has occurred and carrying out measures such as shutting down certain ports, may result in unjustifiable denial of service to legitimate network traffic. In such a case, an event driven framework for automatic decision making under uncertainty is required.

In sensor network applications, a large number of sensors continuously transmit relatively rudimentary data (in the form of events) to some central server. An example may be a monitoring application, in which sensors throughout a large office building constantly transmit the temperature of each room in the building. An application may then be required to detect the possibility of a fire occurring in the building based on the individual readings or to aggregate the individual readings.

Because of current technical limitations, each individual sensor is unreliable. Therefore, it is quite possible that a temperature reading is not transmitted by the sensor or that there is an error in the transmitted reading. Due to this unreliability, an explicit treatment of uncertainty is required in such domains.

Future Directions

While research in event-based systems and event driven applications has been ongoing for several years, relatively little research has been carried out on the explicit treatment of uncertainty in events. Moreover, most such research has been focused on specific domains such as sensor networks and security application. Therefore, much work remains in exploring new and formal approaches to such treatment. An additional important future direction is the implementation of a production level system that enables event inference under uncertainty in the general case.

Cross-references

- ▶ Active and Real-time Data Warehousing
- ▶ Atomic Event
- ▶ Complex Event
- ▶ Complex Event Processing
- ▶ Composite Event
- ▶ Event
- ▶ Event and Pattern Detection over Streams
- ▶ Event Driven Architecture
- ▶ Event Prediction
- ▶ Explicit Event
- ▶ Implicit Event

Recommended Reading

1. Balazinska M., Khoussainova N., and Suciu D. PEEX: extracting probabilistic events from rd data. In Proc. 24th Int. Conf. on Data Engineering, 2008.
2. Halpern J.Y. Reasoning About Uncertainty. MIT Press, Cambridge, MA, 2003.
3. Li C.-S., Aggarwal C., Campbell M., Chang Y.-C., Glass G., Iyengar V., Joshi M., Lin C.-Y., Naphade M., and Smith J.R. Epi-spire: a system for environmental and public health activity monitoring. In Proc. IEEE Int. Conf. on Multimedia and Expo., 2003.
4. Luckham D. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Reading, MA, 2002.
5. Patton N.W. Active Rules in Database Systems. Springer, Berlin, 1999.
6. Wasserkrug S., Gal A., and Etzion O. A model for reasoning with uncertain rules in event composition systems. In Proc. 21st Annual Conf. on Uncertainty in Artificial Intelligence, 2005.

Uncertainty Management in Scientific Database Systems

Nilesh Dalvi
Yahoo! Research, Santa Clara, CA, USA

Definition

Scientific databases often deal with data that comes from multiple sources of varying quality, is heterogeneous, incomplete and inconsistent, and ridden with measurement errors. Uncertainty management deals with a set of techniques for modeling and representing the various uncertainties that arise in scientific data and to enable users to query the data. This entry describes the UII system [10] that addresses the issue of managing uncertainty in integrating scientific databases.

Historical Background

Distributed data integration is becoming increasingly popular in biomedical research and in scientific

research in general. Its popularity is based on the realization that combining sources frequently lead to novel scientific discoveries that cannot be concluded from any single source in isolation. However, as more and more scientific data is shared and as tools are built to provide a common query interface for them, the scientists face the major problem of dealing with information overload [19]. There are several factors that affect the quality of answers to the user queries: the quality of the data sources themselves, the quality of links across data sources, the alignment of data across sources, and so on. The data sources have a large variation in the quality of their data, caused by the curation method, provenance and the experimental protocols used in data collection. Some databases are nicely linked, with one containing foreign keys to other, while others have to be linked by matching tuples over text fields. The scientific data across sources is often not aligned, due to experimental errors and lack of common standards. Specialized algorithms exist to align scientific data, e.g., the BLAST [16] sequence alignment algorithm for aligning potentially similar proteins, but these algorithms are speculative at best. As a consequence, the user is left with the daunting task of searching for relevant answers among a vast number of spurious results.

Uncertain data management [15] has seen a recent renewed interest in the databases community. This is because of the thrust from applications that span not just scientific databases but several others that include exploratory queries in databases, novel IR-style approaches to data integration, information extraction from Web, sensor networks applications and data privacy analysis. There are efforts to build general purpose uncertain data management systems, e.g., MystiQ [2] and Trio [18]. In parallel, there are several efforts in the community to develop scientific data management systems and tools to ease the tasks of scientists and facilitating the process of sharing and reusing experimental data. This entry describes the application of uncertain data management techniques to scientific databases. It gives several examples of specific uncertainties that arise in managing scientific data and describes the UII system [10] that addresses these issues.

Foundations

This section provides examples of data uncertainties that arise in scientific data.

Uncertainties in Scientific Data

The uncertainties in scientific data can be classified into two broad categories:

1. *Inherent Data Uncertainties* Inherent data uncertainties are attributes of the data itself and not artifacts of its representation. Data generated from laboratory experimental methods often have inherent uncertainties. To illustrate an extreme case, two-hybrid screening assays, which are used to detect protein interactions, have error rates estimated to be close to 50% [2]. Experimental data can also be generated from computational experiments. The BLAST algorithm [3] searches in a database for sequences similar to a query sequence. The similarity between any two sequences is measured by the BLAST evalue, which is the degree to which the pairing could occur by chance. Additionally, uncertainties can be rooted in the ever-evolving nature of biological knowledge itself. For example, GenBank references sequences (RefSeqs) are assigned status codes which refer to the amount of evidence and expert curation attributed to a given sequence and its function [4]. These codes range from "inferred" where there is little support for a given sequence, to "reviewed" where substantial evidence exists and has been vetted by a biological domain expert. Status codes for sequences change over time as evidence for them accumulates.
2. *Data Representation Uncertainties* Data representation uncertainties result from the mapping of real world information onto a computable representation of this information. At last count there were over 600 online data sources in molecular biology [5]. Unfortunately, for all the data that is available there are no common standards for representing it (in part due to the evolving nature of biomedical knowledge). The result is the decentralized and heterogeneous nature of biological data sources which is an underlying source of many data uncertainties. For instance, there is no common identifier for a biological object [6] which make it difficult to query across data sources (manually or otherwise), a task which is commonly performed. Linkages between data records may then require string matches on text fields rather than more reliable "foreign-key" relationships. Additionally, data sources tend to represent data in idiosyncratic fashion. For

example, GenBank uses RefSeq status codes to represent the level of evidence for a particular gene but the Gene Ontology (GO) uses evidence codes [7]. Given evidence from both sources, it is sometimes difficult to make comparisons, such as determining which code provides the greater weight of evidence.

General Purpose Systems for Managing Uncertainty

The problem of managing uncertainty in databases has a long history [1,3,6,9] with a recent renewed interest, and several systems for managing uncertain data have been proposed recently in the literature.

Trio The Trio [18] system being developed at Stanford is a data management system that supports uncertainty and lineage. The focus of the system is on the representation formalisms for uncertainty and lineage as well as query languages over such data. Trio extends the relational database model with (i) *alternative values*, where tuple attributes may be assigned a set of possible values rather than a single value, (ii) *maybe* ("?") annotations, specifying that a tuple may not exist, (iii) optional *numeric confidences* attached to alternatives and (iv) *lineage*, connecting tuple-alternatives to other tuple alternatives from which they were delivered. The Trio query language extends SQL to support querying based on tuple-alternatives as well as lineage.

MystiQ The MystiQ [2] system from University of Washington is another general-purpose system which supports various constructs for handling uncertainty that include probabilities associated with tuples, approximate predicates in SQL queries and *soft views* over uncertain data. The focus of the system is on efficient query evaluation, and uses various techniques like *safe query plans* [4] and Monte-carlo approximations [11].

Other Systems The Orion system [14] at Purdue looks at uncertainty given by continuous-valued probability distributions. This is an important problem which is specially relevant to managing uncertainty in sensor networks, where the actual sensor readings often have continuous-valued uncertainty associated with it. The work looks at supporting aggregate queries over such data items. There is also very interesting research [5,7,12] being carried out on extending probabilistic databases to represent correlated tuples and graphical models.

Similarly, there are several systems that exist for Scientific data management. However, several challenges must be met to combine the uncertain data management techniques with these systems. First, it requires an analysis of various sources of uncertainties that arise in Scientific data management, and a principled way to quantify and represent these uncertainties. Secondly, the uncertain data management systems primarily use SQL queries over relational data. This model is unsuitable [13] for scientific databases as the scientists using the system are unfamiliar with the concepts of databases and SQL queries and prefer a simpler interface to browse and explore the data. The rest of this entry describes the UII system[10] which is a specialized scientific data management system with support for uncertainty.

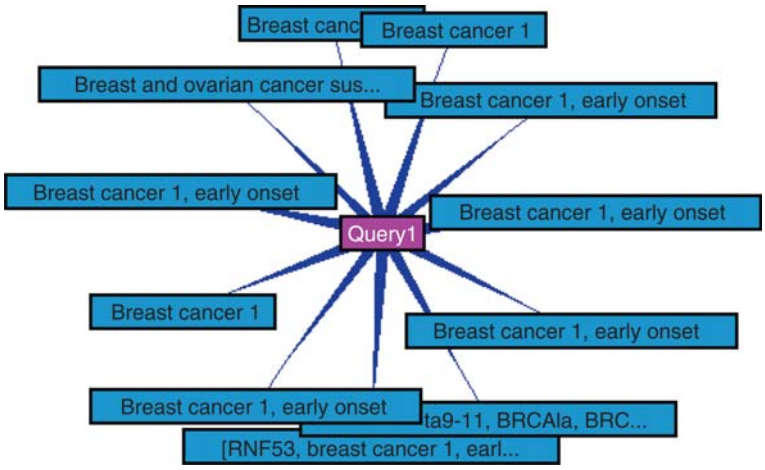
The UII System for Managing Uncertainty in Scientific Data

UII is built on top of the BioMediator system [13,17], which is a mediated-schema distributed data integration system being developed at the University of Washington. The query model of BioMediator is designed to address the need of biologists, which supports poorly specified, exploratory kind of queries.

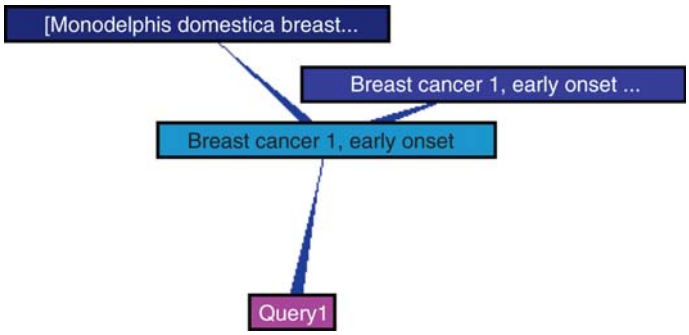
A user begins by issuing a simple query, called the *seed query*, which establishes the basic topic of interest. The user can then browse these results and explore any of the results further by finding information about it in other sources, a process called *query expansion*.

Figure 1 shows a sample seed query: `Gene` records containing a `Symbol` attribute with value `BRCA1`. The system determines which of the included data sources contain `Gene` information and then queries these sources for entries satisfying the specified constraints. The result is shown as a graph that contains one query node and several results node linked to it.

A user can expand a set of node by clicking them, which causes them to join with tuples from other data sources, using the information present in the nodes to be expanded. Such information may include foreign keys to other sources allowing direct look-ups. Often, in the absence of unique foreign keys, the joins involve partial matches over (multiple) free text fields. Figure 2 shows an example of a single step of query expansion. The one light node is expanded once resulting in two dark nodes.



Uncertainty Management in Scientific Database Systems. Figure 1. A seed query.



Uncertainty Management in Scientific Database Systems. Figure 2. A single step in query expansion.

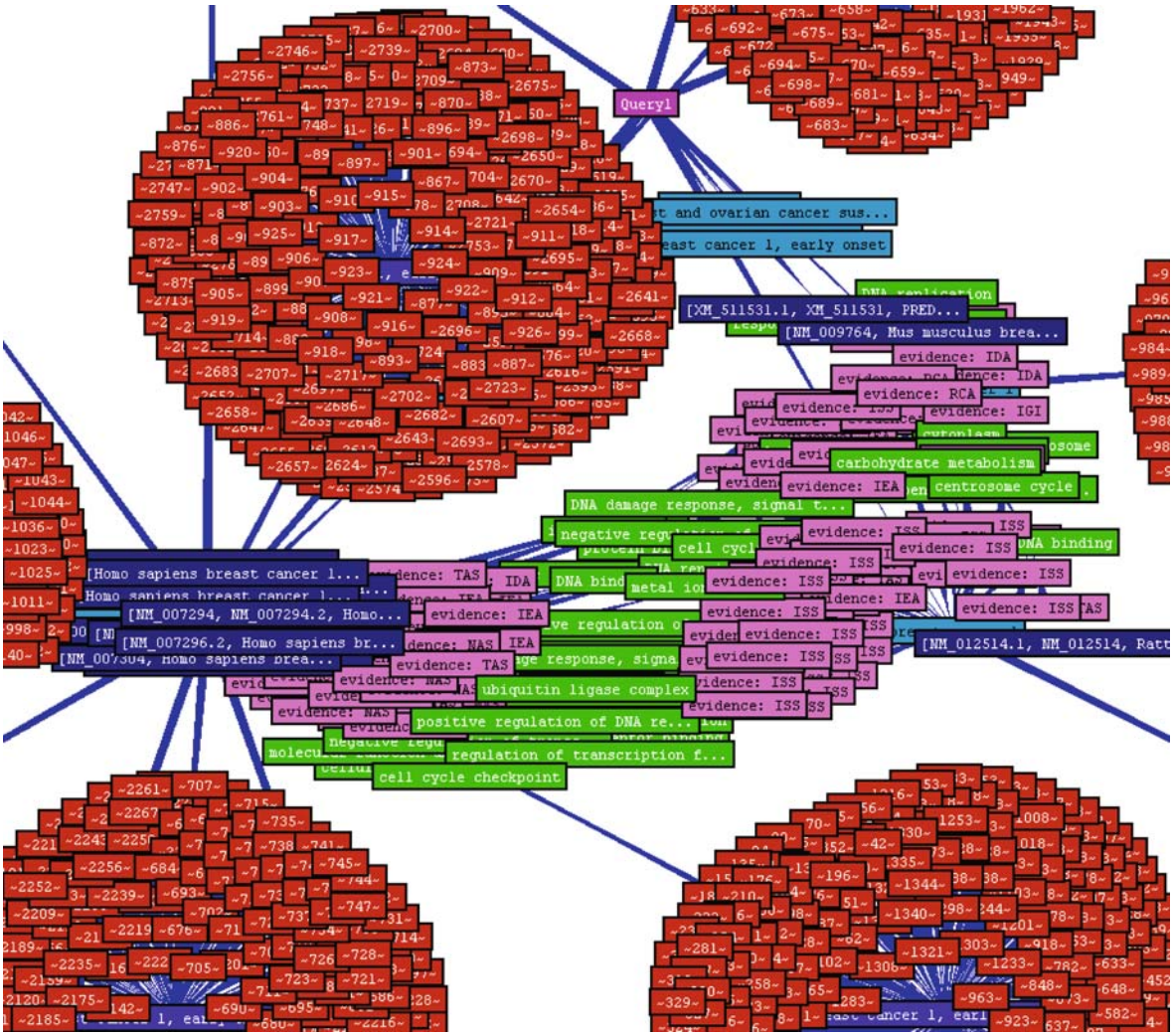
The browsing history of an user is abstracted in terms of a *browsing graph*. A browsing graph is a directed graph whose nodes are records from data sources, and edges correspond to node expansions. It has a distinguished query node, which in the seed query used to initiate the browsing. There can be multiple paths from the start node to a give node, which corresponding to multiple expansion paths to the same node.

Handling Uncertainty

The problem of searching for relevant query answers is exemplified by Fig. 3, which shows a fragment of browsing graph with over 5,000 results, derived by just two complete expansion steps (where in each step all the nodes are expanded) starting from the same seed query shown in Fig. 1. However, not all the nodes in the browsing graph are equally likely to be useful to the user. A node may be of low quality because of the data source that contributes that node. An expansion edge may be of low quality because it was joined using a partial match on a text field. The farther a node is from the start node, the less likely it is to be relevant to the user. The relevance of nodes to the user is assessed in two steps. In the first step, the browsing graph is annotated with *uncertainty metrics*, that describe the quality of each node and each edge in the graph. In the second step, these metrics are used to compute the relevance of each node to the user. The two steps are described below.

Uncertainty Metrics The following is a summary description of the four fundamental uncertainty metrics that capture all types of uncertainty in the UII system:

Ps Measure: Ps is a measure defined at the schema level. It is a quantification of user's prior belief in the quality of data records of a particular data type from a particular data source (e.g., Genes from Entrez Gene, Classifications from Entrez Gene, and Classifications from GO, are each assigned Ps values). For example, consider the comparison between proteins from



Uncertainty Management in Scientific Database Systems. Figure 3. A small portion of a large BioMediator result set. The seed query plus two subsequent expansions produced over 5,000 results.

SwissProt and TrEMBL. SwissProt is a manually and carefully curated data source of protein functional information whereas TrEMBL contains only computational predictions which are deemed less reliable. The class of protein records from SwissProt therefore are assigned a higher Ps value than those from TrEMBL because SwissProt protein records are generally trusted to a greater degree.

Qs Measure: Qs is also defined at the schema level. It is a quantification of user's prior belief in the quality of links between to data types in two different data sources (e.g., Genes in Entrez Gene to Proteins in Entrez Protein). To elaborate, records in some sources, such as Gene records from Entrez Gene, contain references to records in another source, such as Protein records from Entrez Protein. In this example, these references are in the form of "accession" numbers which essentially correspond to unique identifiers (foreign keys). Records between other types and sources however may only be connected by non-foreign keys, e.g., text-string similarities such as is the case between Genes from Entrez Gene and Genes in OMIM. In this example, the relationship between Genes in Entrez Gene and Proteins in Entrez Protein is assigned a higher Qs value since these links are better in general than those between Genes in Entrez Gene and Genes in OMIM.

Pr Measure: This is defined at data level and is a quantification of user's belief in a particular data record. It is used to capture data uncertainties which differ between records of the same type and source.

Gene records in Entrez Gene, for example, are attributed with a Refseq status code which ranges in value from "inferred" to "reviewed." These status codes correspond to the amount of evidence for a given gene, therefore "reviewed" are assigned a higher Pr value than "inferred."

Qr Measure: Qr is also defined at the data level, and is a quantification of user's belief in a particular cross-reference (link) between two data records. It is dynamic (calculated at the time two linked results are returned by the system). For example, record cross-references using unique identifiers always receive a Qr of 1.0. Some records may reference others via the use of comparison algorithms such as BLAST. For BLAST cross-references, Qr scores are dynamically computed by converting the e-value from the BLAST algorithm into a numeric value between 0.0 and 1.0. BLAST comparisons that correspond to better matches between records (higher similarity) receive higher Qr values.

Probabilistic Query Evaluation Each query in UII system results in a set of nodes forming the browsing graph for the query. The objective is to find the relevance of each node in the browsing graph to the original seed query. This can be posed as a network reliability problem [8]. For each node in the graph, the quantity $Ps * Pr$ is interpreted as the probability that the node is correct. Similarly for each edge, $Qr * Qs$ is interpreted as the score that the edge is correct. Finally, the relevance of a node is simply the probability that the node is reachable from the seed query node in the browsing graph.

The exact relevance computation is an intractable problem [8]. However, the probabilities can be efficiently approximated to arbitrary precision using simulation algorithms. The following simulation algorithm can be employed, which is well suited for the needs of the problem. In a single pass, N trials (path traversals in the graph) are simulated where nodes and edges are included in the traversal with their associated probabilities. This is done by first storing a random N-bit vector with each node/edge, that denotes whether the node is in or out in the corresponding experiment. Finally, performing a single depth-first search and using these bit vectors, a final bit vector for each node is obtained that describes the trials where that node was reachable from the start node. The relevance for a node is then estimated using the quantity k/N, where k is the number of set bits in the node's final bit vector. The choice of N influences the error in the estimation, the larger the N the smaller the error. Also, for any fixed value of N, the larger the actual relevance of a node, the better the approximation. Thus, the simulation will correctly rank the most relevant answers, which the user cares about most, while the poorest results may be slightly out of order.

Key Applications

Uncertainty management techniques are fundamental to *Scientific Data Integration* applications.

Future Directions

Uncertainty is a fundamental issue in scientific databases, and there are several open and challenging problems which need to be addressed. One important problem is to assign meaningful probability scores to data in a principled way. While the current approach requires domain experts to specify Ps, Pr, Qs and Qr values, it would be more desirable to automatically ascribe them using machine learning techniques. Uncertainty management systems should also be able to relearn and update these scores based on users' feedback to query results. Another problem is to develop highly scalable and efficient query answering algorithms that go beyond the current simulation based techniques. A potential approach in this direction is to use the uncertainty scores to enable a focused expansion of the query rather than constructing the whole browsing graph a priori and ranking its nodes. Another important aspect of managing uncertainty in scientific data is keeping track of data provenance. As scientists use experimental data from different sources and their own experimental data to generate new data, it becomes important to track the provenance of data through these computing processes. The uncertainties in scientific data are often correlated in ways dictated by their provenance, and managing uncertainty along with the provenance information is a challenging and important future direction.

Cross-references

- ▶ Data Cleaning
- ▶ Data Integration
- ▶ Data Provenence
- ▶ Data Quality Models
- ▶ Inconsistent Databases
- ▶ Probabilistic Databases

- ▶ Provenance of Scientific Databases
- ▶ Record Linkage
- ▶ Record Matching

Recommended Reading

1. Barbará D., Garcia-Molina H., and Porter D. The management of probabilistic data. IEEE Trans. Knowl. Data Eng., 4(5):487–502, 1992.
2. Boulos J., Dalvi N., Mandhani B., Mathur S., Re C., and Suciu D. Mystiq: a system for finding more answers by using probabilities. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 891–893.
3. Cavallo R. and Pittarelli M. The theory of probabilistic databases. In Proc. 13th Int. Conf. on Very Large Data Bases, 1987, pp. 71–81.
4. Dalvi N. and Suciu D. Efficient query evaluation on probabilistic databases. In Proc. 26th Int. Conf. on Very Large Data Bases, 2004, pp. 864–875.
5. Deshpande A. and Sunita Sarawagi. Probabilistic graphical models and their role in databases. In Proc. 33rd Int. Conf. on Very Large Data Bases, 2007, pp. 1435–1436.
6. Dey D. and Sarkar S. A probabilistic relational model and algebra. ACM Trans. Database Syst., 21(3):339–369, 1996.
7. Garofalakis M.N., Brown K.P., Franklin M.J., Hellerstein J.M., Wang D.Z., Michelakis E., Tancau L., Wu E., Jeffery S.R., and Aipperspach R. Probabilistic data management for pervasive computing: The data furnace project. IEEE Data Eng. Bull., 29(1):57–63, 2006.
8. Karger D.R. A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. In Proc. 27th Annual ACM Symp. on Theory of Computing, 1995, pp. 11–17.
9. Lakshmanan L.V.S., Leone N., Ross R., and Subrahmanian V.S. Probview: a flexible probabilistic database system. ACM Trans. Database Syst., 22(3):419–469, 1997.
10. Louie B., Detwiler L., Dalvi N., Shaker R., Tarczy-Hornoch P., and Suciu D. Incorporating uncertainty metrics into a general-purpose data integration system. In Proc. 19th Int. Conf. on Scientific and Statistical Database Management, 2007, pp. 19–28.
11. Re C., Dalvi N., and Suciu D. Efficient top-k query evaluation on probabilistic data. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 886–895.
12. Sen P. and Deshpande A. Representing and querying correlated tuples in probabilistic databases. In Proc. 23rd Int. Conf. on Data Engineering, 2007, pp. 596–605.
13. Shaker R., Mork P., Brockenbrough J.S., Donelson L., and Tarczy-Hornoch P. The biomediator system as a tool for integrating biologic databases on the web. In Proc. Workshop on Information Integration on the Web, 2004.
14. Singh S., Mayfield C., Mittal S., Prabhakar S., Hambrusch S., and Shah R. Orion 2.0: native support for uncertain data. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2008, pp. 1239–1242.
15. Suciu D. and Dalvi N. Foundations of probabilistic answers to queries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 963.
16. Tatusova T.A. and Madden T.L. Blast 2 sequences – a new tool for comparing protein and nucleotide sequences. FEMS Microbiol. Lett., 174:247–250, 1999.
17. Wang K., Tarczy-Hornoch P., Shaker R., Mork P., and Brinkley J. Biomediator data integration: Beyond genomics to neuroscience data. In AMIA Fall 2005 Symposium Proceedings, 2005, pp. 779–783.
18. Widom J. Trio: a system for integrated management of data, accuracy, and lineage. In Proc. 2nd Biennial Conf. on Innovative Data Systems Research, 2005.
19. Woods D.D., Patterson E.S., Roth E.M., and Christoffersen K. Can we ever escape from data overload? a cognitive systems diagnosis. Cogn. Technol. Work, 4(1):22–36, 2002.

Undo

- ▶ Logging and Recovery

Unicode

Ethan V. Munson
University of Wisconsin-Milwaukee, Milwaukee, WI, USA

Definition

Unicode is an international standard for representing text characters. Unicode supports the scripts of most languages in wide use and has a flexible design that is capable of supporting all known human languages and all of their variant scripts. The development of the Unicode standard is coordinated by the Unicode Consortium.

Key Points

Unicode's development is motivated by the need to encode characters in all languages without conflicts between the encodings for different languages. Obviously, the achievement of this goal is fraught with technical and political complexities.

Unicode has several different encodings. The most widely used is the 8-bit, variable-width UTF-8 encoding, which permits the encoding of many European languages in an efficient one-byte form and is backward compatible with both the ASCII and ISO-8859-1 character sets. UTF-16 is a 16-bit, variable-width encoding that is more suitable to languages with many characters, such as Chinese, Japanese and Korean. UTF-32 is a

four-byte, fixed-width encoding that encompasses all Unicode characters.

The Unicode Consortium is supported by many well-known computer and software manufacturers. For the members of the consortium, the system internationalization problem is of compelling importance.

Cross-references

- ▶ Document
- ▶ Document Representations (Inclusive Native and Relational)
- ▶ Unified Modeling Language (UML)
- ▶ XML

Unified Modeling Language

Martin Gogolla
University of Bremen, Bremen, Germany

Synonyms

UML; Unified modeling language; Unified modeling language

Definition

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [8].

Historical Background

The UML is based on earlier software design approaches, among them the Object Modeling Technique (OMT) [9], Object-Oriented Analysis and Design (OOAD) [1], and Object-Oriented Software Engineering (OOSE) [6] and other important techniques [5]. The UML was standardized in various versions by the Object Modeling Group (OMG) [8]. Authors of the predecessor techniques have published a comprehensive description of the UML [10]. Questions strongly related to UML are discussed at the UML and MODELS conferences [13]. Special issues with extended papers from that conference and other works strongly related to UML and modeling are published in [11].

UML is mainly a graphical language and offers different diagram types. Among the most important diagram types are the class, object, statechart, activity, sequence, communication, and use case diagram. UML includes the Object Constraint Language (OCL) which is a textual language for navigation in class diagrams and for expressing textual constraints like invariants, pre- and postconditions, or guard conditions. Many tools for UML, but fewer support for OCL is available, see e.g., [3,12], among other interesting work.

Foundations

UML Class and Object Diagrams

The main purpose of class diagrams is to capture the static structures and operations of a system. This section briefly explains the most fundamental features in class diagrams: classes and associations. More advanced features are also discussed.

Classes: A class is a descriptor for a set of objects sharing the same structure and behavior. In each (database) system state, a class is manifested by a set of objects. In the database context, concentration is upon structural aspects, although behavioral aspects may be represented in UML as well. Object properties can be described by attributes classified by data types like `String` or `Boolean`. Properties can also stem from roles in associations which connect classes.

Example: Figure 1 shows the classes `Supplier`, `Project`, and `Part` together with some basic attributes including their data types, e.g., one identifies `Supplier::Name:String` and `Project::Budget:Integer`. In this contribution, the general scheme for denoting properties (attributes and roles) is `Class::Property:PropertyType`.

Associations: An association represents a connection among a collection of classes and may be given a name. An association is manifested by a set of object connections, so-called links, sharing the same structure. A binary association can be defined between two different classes; objects of the respective classes play a particular role in the association. A binary association can also be defined on a single class; then objects of the class can play two different roles; such a binary association is called reflexive. A ternary association involves three roles. The notion n-ary association refers to a ternary or a higher-order

association. Binary associations are usually shown with a simple line, and an n-ary association with a small rhomb-shaped polygon.

Example: Fig. 1 shows the binary association `ProjectPart` with roles `project` and `part`, the ternary association `SupplierProjectPart` with roles `supplier`, `suppliedProject`, and `suppliedPart`, and the reflexive association `Component` with roles `parent` and `child`.

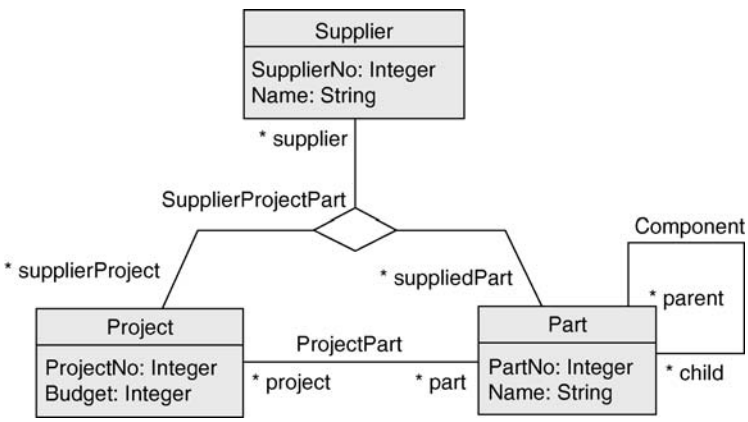
Objects and Links: Structural aspects in UML can also be represented in an object diagram showing objects, links, and attribute values as manifestations of classes, associations, and attributes. An object diagram shows an instantiation of a class diagram and represents the described system in a particular state. Underlining for objects and links is used in object diagrams in order to distinguish them clearly from class diagrams.

Example: Figure 2 shows an object diagram for the class diagram from Fig. 1. Objects, links, and attribute values correspond to the classes, associations, and attributes. There are two `Project` objects, two `Supplier` objects and five `Part` objects. Each `Part` object represents a piece of a software controller (`Ctrl`) being responsible for a particular portion of a car. The `Component` links express part-of relationships, for example, the `Engine Code (engineCtrl)` includes the `Battery Code (batteryCtrl)` and the `Motor Code (motorCtrl)`.

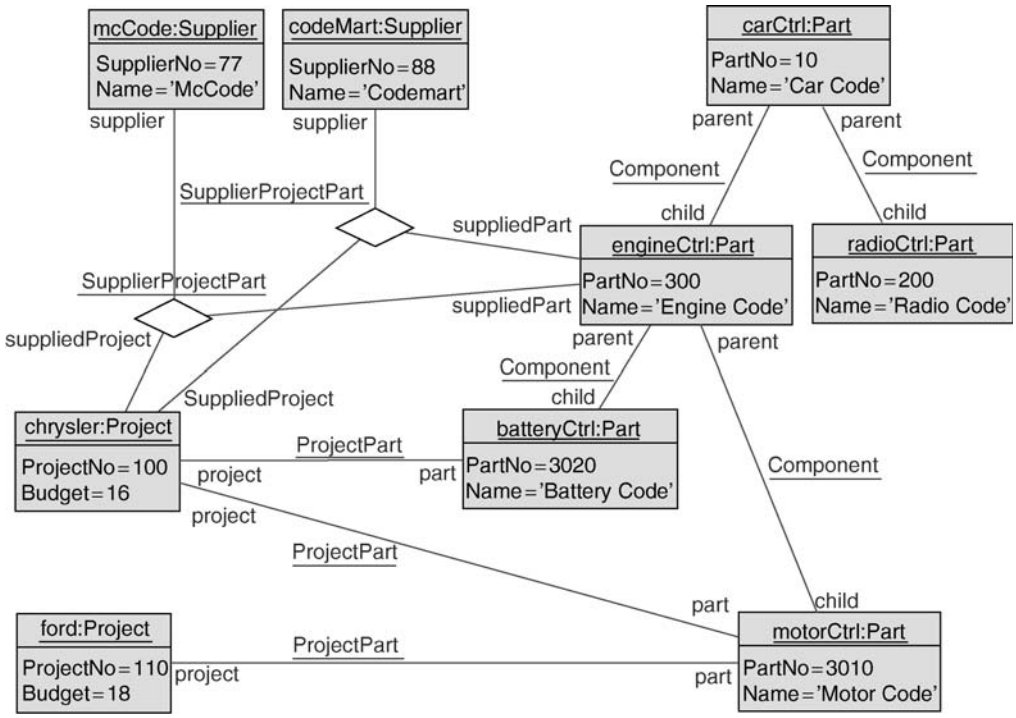
Roles: Proper roles must be specified on a class diagram in order to guarantee unique navigation. Navigation in a class diagram means to fix two classes and to consider a path from the first class to the second class by using association roles. The roles on the opposite side of a given class in an association also determine properties of the given class by navigating via the roles. Therefore, in UML the opposite side roles must be unique. Recall that properties can also come from attributes.

Example: Roles are captured on links. This is necessary in reflexive associations and in other situations, for example, if two associations are present between two given classes. For example in Fig. 2, if one considers the link between `carCtrl` and `engineCrtl`, without roles one could not tell which object plays the `parent` role and which one the `child` role. In the class diagram in Fig. 1, the class `Project` has two direct navigation possibilities to class `Part`: One via association `ProjectPart` and the other one via association `SupplierProjectPart`. One obtains therefore two properties of class `Project` returning `Part` objects: `Project::part:Set(Part)` from association `ProjectPart` and `Project::suppliedPart: Set(Part)` from association `SupplierProjectPart`. In the object diagram one obtains, for example, `ford.part = Set{motorCtrl}` as well as `ford.suppliedPart = Set{}`. In order to distinguish between these two navigations, the role name `part` and `suppliedPart` must be distinct.

Class Diagram versus Database Schema: In the database context, it is interesting to remark that the connection between a class diagram and its object diagrams resembles the connection between a database schema and its associated database states. The class diagram in general induces a set of object diagrams and the database schema determines a set of database states. Object diagrams and database states follow the general principles formulated in the class diagram and database schema, respectively. Because example object diagrams have to be displayed on a screen or on paper,



Unified Modeling Language. Figure 1. Example UML class diagram 1.



Unified Modeling Language. Figure 2. Example object diagram 1.

they tend to show less information than proper, large database states. They may however explain the principles underlying a class diagram pretty well if the examples are well chosen.

UML Class Diagram Features for Conceptual Schemas

Some of the more advanced features of conceptual database schemas in UML class diagrams are: object-valued, collection-valued and compound attributes, role multiplicities, association classes, generalizations, aggregations, compositions, and invariants.

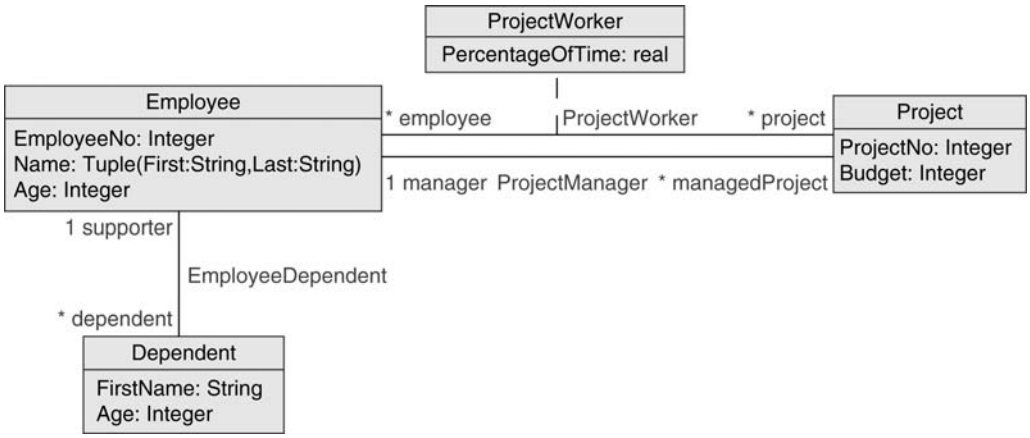
Object-Valued Attributes: Attributes in UML may not only be data-valued as above, but the attribute type may be a class as well which leads to object-valued attributes. Like associations, object-valued attributes also establish a connection between classes. However, an object-valued attribute is only available in the class in which it is defined. The information from that attribute is not directly present in the attribute type class. Thus an object-valued attribute may be regarded as a unidirectional association without an explicit name and where only one role is available.

Examples: The class diagram in Fig. 3 extends the class diagram in Fig. 1 by introducing the new classes `Employee`, `Dependent`, and `ProjectWorker`, and the associations `EmployeeDependent`, `ProjectManager`, and `ProjectWorker`. The fact that `ProjectWorker` is mentioned as a class as well as an association will be explained below. The object diagram in Fig. 4 shows an example state for the class diagrams from Fig. 3.

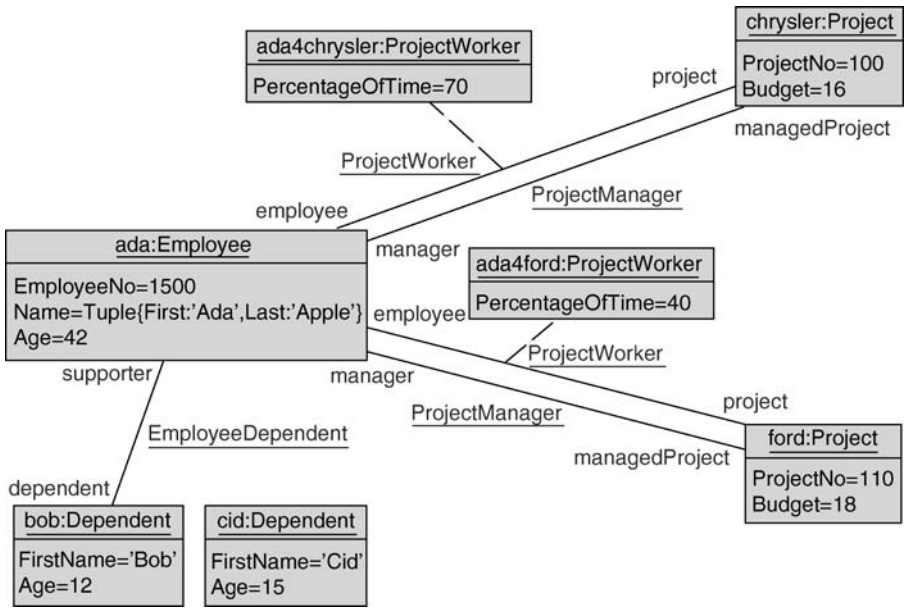
As an example for an object-valued attribute and as an alternative for the association `ProjectManager`, one could extend the class `Project` by an attribute `manager` with type `Employee`. This could be represented altogether as `Project::manager:Employee`.

Collection-Valued Attributes: The collection kinds set, bag, and sequence have already been introduced. These collection kinds can be used as type constructors on data types and classes. For building attribute types, the constructors may be nested.

Examples: An attribute could possess a type like `Set(Project)`. As an alternative for the association `ProjectManager` one could have one attribute `managedProject:Set(Project)` in the class `Employee` and another attribute `manager:Employee` in class `Project`. There is however a significant difference between the model with the association `ProjectManager` including the roles `manager` and



Unified Modeling Language. Figure 3. Example UML class diagram 2.



Unified Modeling Language. Figure 4. Example UML object diagram 2.

`managedProject` and the model with the two attributes `manager` and `managedProject`. In the model with the association, the roles `managedProject` and `manager` represent the same set of object connections, i.e., are inverse to each other:

```
Employee.allInstances->forAll(e|e.
managedProject->forAll(p|p.manager=e))
Project.allInstances->forAll(p|p.manag-
er.managedProject->includes(p))
```

This is not required to hold in the model possessing the two attributes. In this case the two attributes `managedProject` and `manager` are independent from each other and may represent different sets of object connections.

Another useful application of collection-valued types are collections over the data types like the complex type `Set(Sequence(String))`. A value for an attribute typed in this way could be, for example, the complex value `Set{Sequence{'Rome','Euro'}, Sequence{'Tokyo','Yen'}}`.

Compound Attributes: Apart from using the collection constructors `Set`, `Bag`, and `Sequence` for attributes, one can employ a tuple constructor `Tuple`. A tuple has a set of components each possessing a

component discriminator and a component type. The collection constructors and the tuple constructor may be nested in an orthogonal way.

Examples: The above value for the type `Set(Sequence(String))` could be represented also with type `Set(Tuple(Town:String,Currency:String))` and with the corresponding value `Set{Tuple{Town:'Rome', Currency:'Euro'}, Tuple{Town:'Tokyo', Currency:'Yen'}}`.

As a further example of a compound attribute using the `Tuple` constructor, one identifies in the class diagram in Fig. 3 the attribute `Name` in class `Employee` which is a compound attribute with type `Tuple(First:String, Last:String)`.

Role Multiplicities: Associations may be restricted by specifying multiplicities. In a binary association, the multiplicity on the other side of a given class restricts the number of objects of the other class to which a given object may be connected. In a simple form, the multiplicity is given as an integer interval `low..high` (with `low` $\leq$`high`) which expresses that every object of the given class must be connected to at least `low` objects and at most `high` objects of the opposite class. The `high` specification may be given as `*` indicating no higher bound. A single integer `i` denotes the interval `i..i` and `*` is short for `0..*`. The multiplicity specification may consist of more than one interval.

Examples: The multiplicity `1` on the role `supporter` indicates that an object of class `Dependent` must be linked to exactly one object of class `Employee` via the association `EmployeeDependent`.

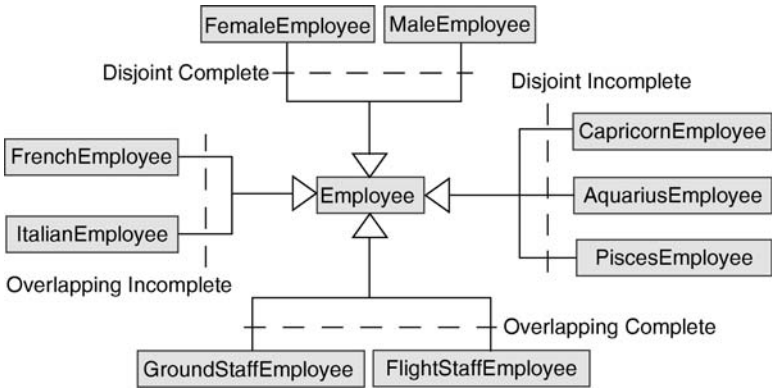
Association Classes: Associations may be viewed again as classes leading to the concept of an association class. Association classes are shown with a class rectangle and are connected to the association (represented by a line or a rhomb) with a dashed line. Association classes open the possibility of assigning attributes to associations.

Examples: The association `ProjectWorker` is modeled also as a class: `ProjectWorker` is an association class. This makes it possible to assign the attribute `PercentageOfTime` to the association `ProjectWorker`. In the class diagram, `ProjectWorker` is redundant as both the class name and the association name; the specification as the class name would be sufficient.

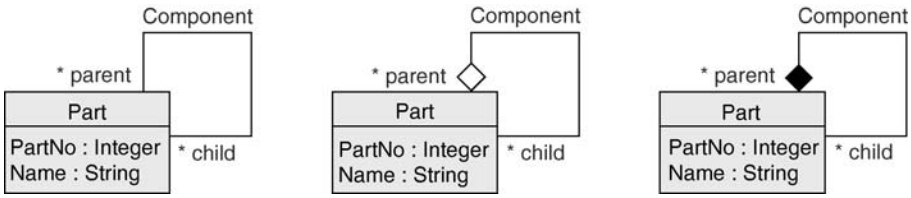
Generalizations: Generalizations are represented in UML with directed lines having an unfilled small triangle pointing to the more general class. Usually the more specific class inherits the properties from the more general class. Generalizations are known in the database context also as ISA (IS-A) hierarchies. In the programming language context often the notion of inheritance shows up. Viewed from the more general class its more specific classes are its specializations. In general, a class may have many specializations, and a class may have many generalizations. A set of generalizations may be restricted to be `disjoint` and a set of generalizations may be classified as `complete`. The classification `disjoint` means that any two specialized classes are not allowed to have a common instance. The label `complete` means that every instance of the general class is also an instance of at least one more specialized class. The explicit keywords `overlapping` and `incomplete` may be attached to sets of generalizations for which no respective restriction is made.

Examples: Figure 5 shows different specializations of the class `Employee`. The subclasses `FemaleEmployee` and `MaleEmployee` represent a disjoint and complete classification. The subclasses `CapricornEmployee`, `AquariusEmployee`, and `PiscesEmployee` classify employees according to their birthday (December 22–January 20, January 21–February 19, February 20–March 20, respectively). This classification is disjoint but incomplete. The subclasses `GroundStaffEmployee` and `FlightStaffEmployee`in the context of an airline company are labeled overlapping and complete, because each airline employee either works on the ground or during a flight and, for example, a flight accident is allowed to work on the ground during boarding and of course during the flight. The subclasses `FrenchEmployee` and `ItalianEmployee` are overlapping because employees may have two citizenships, but it is incomplete because, e.g., Swiss employees are not taken into account.

Aggregations: Part-whole relationships are available in UML class diagrams in two forms [6]. The first form represents a loose binding between the part and the whole, the second form realizes a stronger binding. Both forms can be understood as binary associations with additional restrictions. The first form called aggregation is drawn with a hollow rhomb on the whole side and is often called white diamond. The second form called composition is drawn with a filled rhomb on the whole side and is often called black



Unified Modeling Language. Figure 5. Different example generalizations and specializations in UML.



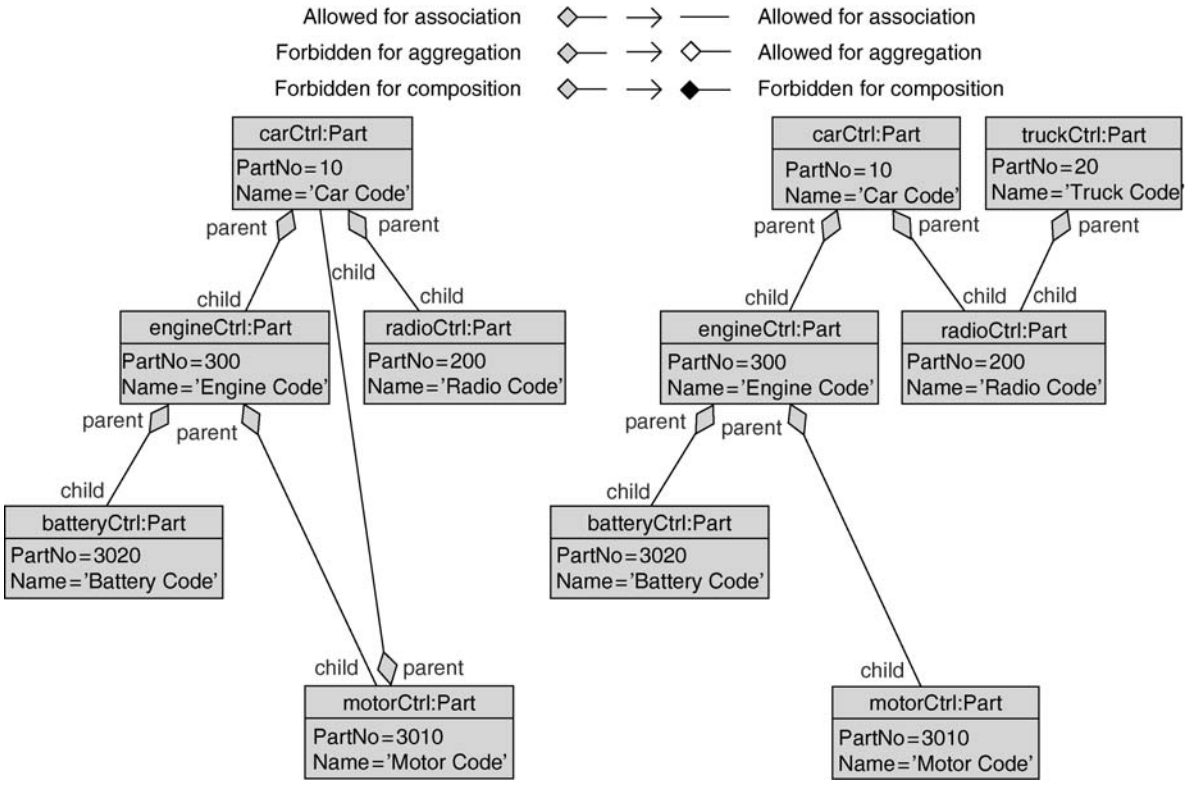
Unified Modeling Language. Figure 6. Component as association, aggregation, and composition.

diamond. The links in an object diagram belonging to a class diagram with a part-whole relationship must be acyclic if one regards the links as directed edges going from the whole to the part. This embodies the idea that no part can include itself as a subpart. Such cyclic links are allowed however for arbitrary associations. Part objects from an aggregation are allowed to be shared by two whole objects whereas this is forbidden for composition.

Examples: The class diagrams in Fig. 6 show on the left the association `Component` already introduced and on the right two alternatives in which the association is classified as an aggregation with a white diamond and as a composition with a black diamond. Recall that roles are essential in reflexive associations and therefore in reflexive part-whole relationships. Here the `parent` objects play the whole role and the `child` objects play the part role. The two object diagrams in Fig. 7 explain the differences between association, aggregation, and composition. The diamonds are shown as grey diamonds, a symbol which does *not* exist in the UML. If the grey diamond is substituted by a white diamond, the left object diagram is forbidden, because there is a cycle in the part-whole links which would mean that the object `carCtrl` is a part of itself. This would also hold for the other two objects on the cycle. Recall that if one would have a simple association instead of the grey diamond, this object diagram would be allowed. If the grey diamond is replaced by a white diamond, the right object diagram is an allowed object diagram. Here, the object `radioCtrl` is shared by the objects `carCtrl` and `truckCtrl`. Naturally, if the grey diamond would become an association, the right object diagram is allowed as well.

Compositions: Compositions pose further restrictions on the possible links in addition to the required acyclicity. Part objects from a composition cannot be shared by two whole objects. The table in Fig. 8 gives an overview on the properties of associations, aggregations, and compositions.

Examples: The discussion now turns to what happens in Fig. 7 if the grey diamond is substituted by a black diamond in order to represent compositions. If the grey diamond is replaced by a black diamond, the left object diagram is again forbidden, because there is a cycle in the part-whole links. If the grey diamond is replaced by a black diamond, the right object diagram is a forbidden object diagram for compositions, because sharing of objects is not allowed in that case.



Unified Modeling Language. Figure 7. Forbidden and allowed object diagrams for aggregation and composition.

To show also a positive example for composition and aggregation, if the link from `motorCtrl` to `carCtrl` is removed in the left object diagram, it is a valid object diagram for compositions and aggregations.

Data Types And Enumeration Types: UML offers a collection of predefined data types with the usual operations on them. The data types include `Integer`, `Real`, `String`, and `Boolean`. Application dependent enumeration types can also be defined in a class diagram. The enumeration type name is followed by the list of allowed enumeration literals. Enumeration types can be used as attribute, operation-parameter or operation-return types.

Examples: Figure 9 shows two enumeration types useful in the context of the running example. The type `Gender` may represent the gender of an employee and the type `CivilStatus` its civil status.

Invariants: OCL allows invariants to be specified, i.e., conditions which must be true during the complete lifetime of an object (or perhaps more precisely, at least, in moments when no activity in the object takes place). Such invariants are implicitly or explicitly universally quantified OCL formulas introduced with the keyword `context`.

	Acyclicity	Prohibition of sharing
Association	–	–
Aggregation	+	–
Composition	+	+

Unified Modeling Language. Figure 8. Overview on properties of associations, aggregations and compositions.

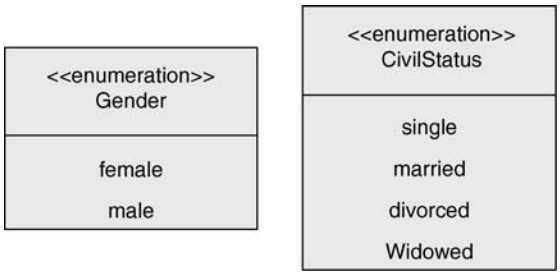
Example: In order to require that employees are atleast 18 years of age, one could state the following invariant.

context Employee inv EmployeeAreAtLeast18: Age>=18

This constraint has an implicit variable `self` of type `Employee` and is equivalent to:

context self:Employee inv EmployeeAreAtLeast18: self.Age>=18

Instead of `self` one could have used any other name for the variable, e.g., the variable `e`. The invariant



Unified Modeling Language. Figure 9. Enumerations in UML.

corresponds to the following OCL formula which must be true in all system states.

Employee.allInstances->forAll(self|self.age>=18)

Key Applications

UML class diagrams are widely used to describe structural aspects for databases (see [7,14], among other relevant papers). OCL constraints are applied for the specification of database integrity. OCL may also be regarded as an object-oriented navigation and query language.

Cross-references

- ▶ Data Model
- ▶ ER Model
- ▶ Extended Entity-Relatioinship Model
- ▶ Object Constraint Language
- ▶ Specialization and Generalization

Recommended Reading

1. Booch G. Object-Oriented Design with Applications. Benjamin-Cummings, Menlo Park, CA, 1991.
2. Chen P.P. The Entity-Relationship Model – Toward a Unified View of Data. ACM Trans. Database Syst., 1(1):9–36, 1976.
3. Gogolla M., Büttner F., and Richters M. USE: A UML-based Specification Environment for Validating UML and OCL. Sci. Comput. Program., 69:27–34, 2007.
4. Gogolla M. and Richters M. Expressing UML class diagrams properties with OCL. In Advances in Object Modelling with the OCL, T. Clark, J. Warmer (eds.). Springer, Berlin Heidelberg New York, 2001, pp. 86–115.
5. Harel D. Statecharts: a visual formalism for complex systems. Sci. Comput. Program., 8(3):231–274, 1987.
6. Jacobson I., Christenson M., Jonsson P., and Oevergaard G. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, MA, USA, 1992.
7. Marcos E., Vela B., and Cavero J.M. A methodological approach for object-relational database design using UML. Software Syst. Model., 2(1):59–75, 2003.
8. OMG (ed.). OMG Unified Modeling Language Specification. OMG, 2007. www.omg.org.
9. Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W. Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (NJ), 1991.
10. Rumbaugh J., Booch G., and Jacobson I. The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading, MA, USA, 2005.
11. SOSYM Editorial Board. Software and Systems Modeling. Springer, Berlin Heidelberg New York, 2007.
12. Toval J.A., Requena V., and Fernandez J.L. Emerging OCL tools. Software Syst. Model., 2(4):248–261, 2003.
13. UML and MODELS Steering Committee. International ACM/IEEE Conference on Model Driven Engineering Languages and Systems MODELS (previously ACM/IEEE International Conference on the Unified Modeling Language UML). http://www.modelsconference.org/.
14. Urban S.D. and Dietrich S.W. Using UML class diagrams for a comparative analysis of relational, object-oriented, and object-relational database mappings. In Proc. of 34th SIGCSE Technical Symp. on Computer Science Education, 2003, pp. 21–25.

Uniform Resource Identifier

- ▶ Resource Identifier

Union

Cristina Sirangelo
University of Edinburgh, Edinburgh, UK

Synonyms

Union

Definition

The union of two relation instances R_1 and R_2 over the same set of attributes U – denoted by $R_1 \cup R_2$ – is another relation instance over U containing precisely the set of tuples t such that $t \in R_1$ or $t \in R_2$.

Key Points

The union is one of the primitive operators of the relational algebra. It is a natural extension of the set union to relations; the additional restriction is that it can be applied only to relations over the same set of attributes. However the union of two arbitrary relations having the same arity can be obtained by first renaming the attributes of one of the two relations.

As an example, consider a relation *Students* over attributes (*number, name*), containing tuples {(1001, *Black*), (1002, *White*)}, and a relation *Employees* over attributes (*number, name*), containing tuples {(1001, *Black*), (1003, *Brown*)}. Then the union *Students* ∪ *Employees* is a relation over attributes (*number, name*) with tuples {(1001, *Black*), (1002, *White*), (1003, *Brown*)}.

In the absence of attribute names, the union is defined on two relations with the same arity. The output is a relation with the same arity as the input, containing the union of the sets of tuples in the two input relations.

The union operator is commutative and associative. The number of tuples in the output relation is bounded by the sum of the number of tuples in the input relations.

Cross-references

- ▶ Relation
- ▶ Relational Algebra
- ▶ Renaming

Uniqueness Constraint

- ▶ Key

Unnoticability

- ▶ Unobservability

Unobservability

Simone Fischer-Hübner
Karlstad University, Karlstad, Sweden

Synonyms

Unnoticability

Definition

Unobservability ensures that a user may use a resource or service without others, especially third parties, being able to observe that the resource or service is being used [1].

A corresponding, but more general definition is provided by [2]. Unobservability of an item of interest (e.g., a subject, messages, action) means that all uninvolved subjects cannot sufficiently distinguish whether or not it exists. Besides, anonymity of subjects involved in the item of interest is provided even against the other subjects involved in that item of interest.

Key Points

Whereas anonymity and pseudonymity protect the relationship of subjects to other items of interest (e.g., the fact that a specific user has sent a message), unobservability protects information about the very existence of the item of interest against uninvolved parties (e.g., the fact that a message was sent). With respect to the same attacker, if a subject's action is unobservable, then the user is also anonymous (see also [2]).

Cross-references

- ▶ Anonymity
- ▶ Pseudonymity
- ▶ Privacy
- ▶ Privacy-Enhancing Technologies

Recommended Reading

1. Common Criteria Project, Common Criteria for Information Technology Security Evaluation, Version 3.1, Part 2: Security Functional Requirements, www.commoncriteriaportal.org, September, 2006.
2. Pfitzmann A. and Hansen M. Anonymity, Unlinkability, Unobservability, Pseudonymity, and Identity Management – A Consolidated Proposal for Terminology, Version 0.29, available at: http://dud.inf.tu-dresden.de/Anon_Terminology.shtml (accessed on July, 2007).

Unsupervised Learning

- ▶ Cluster and Distance Measure
- ▶ Clustering Overview and Applications

Unsupervised Learning on Document Datasets

- ▶ Document Clustering

Until Changed

► Now in Temporal Databases

Update Propagation in Peer-to-Peer Systems

► Updates and Transactions in Peer-to-Peer Systems

Updates and Transactions in Peer-to-Peer Systems

Zachary Ives
University of Pennsylvania, Philadelphia, PA, USA

Synonyms

Consistency in peer-to-peer systems; Update propagation in peer-to-peer systems

Definition

In recent years, work on peer-to-peer systems has started to consider settings in which data is updated, sometimes in the form of atomic transactions, and sometimes by parties other than the original author. This raises many of the issues related to enforcing consistency using concurrency control or other schemes. While such issues have been addressed in many ways in distributed systems and distributed databases, the challenge in the peer-to-peer context is in performing the tasks cooperatively, and potentially in tolerating some variation among the instances at different nodes.

Historical Background

Early peer-to-peer systems focused on sharing or querying immutable data and/or files. More modern uses of peer-to-peer technology consider settings in which dynamic data and updates are being made in a distributed, autonomous context. A question of significant interest is how to define consistency in this model, while still allowing at least some autonomy among the sites.

Foundations

The database community has only recently begun to consider peer-to-peer architectures for applications beyond query answering, with recent work in peer data exchange [10] and collaborative data sharing [16]. However, a popular model has been that of *distributed stream processing*, in which streams of data (usually interpreted as insertions of new data) are processed in a distributed or peer-to-peer network [1,7]. Additionally, work in the distributed systems community, particularly on file systems, must consider many issues related to distributed replication and consistency with the granularity of updates typically being at the level of the file, or custom to an application. (See the entry on *peer-to-peer storage*.)

The work on updates and transactions in peer-to-peer systems can be classified based on who is allowed to modify it, and how conflicting modifications are resolved. This can be divided into the following categories:

Single Owner/Primary Copy is a setting in which each data item that originates from some source peer p can *only* be modified by (or through) p – i.e., no other peers are allowed to directly modify that data.

Owner-Resolver protocols allow multiple peers to modify the data, and they typically rely on the owner to resolve any conflicts. If resolution is impossible, they "branch" the data into fully independent instances.

Consensus protocols allow multiple peers to modify the data, and some set of nodes works together to determine how to arbitrate for consistency.

Partial Divergence schemes handle conflicts in a way that results in multiple divergent copies of the data, but they operate at a finer level of granularity than divergent replica protocols, and they allow some portions of the data instance to remain shared even after "branching" two instances.

The remainder of this entry provides more detail on the different approaches and their common implementations.

Single-Owner/Primary Copy

In the simplest schemes, each data item is owned by a single source, which may update that data. Many other nodes may replicate the data but may not change it (except, perhaps, by going through the *primary copy* at the owner). This is sometimes referred to as the *single-writer, multiple readers* problem. In this type of scheme, the owner of the data uses a timestamp (logical or physical) to preserve the serial order of updates, or to arbitrate among different verions of the data. Since there is a single owner and a single clock, any node can look at the data and deterministically choose an ordering.

In the database world, peer-to-peer stream processing or filtering systems, e.g., ONYX [7] and PIER [11], can be considered systems in which each data item (tuple or XML tree) has a single owner. When new data arrives on the stream, this appends to or replaces the previous data items from the same source.

P2P file systems have also employed a single-owner scheme that provides very interesting properties. In particular, CFS [5], PAST [8], and similar systems build a filesystem layer over distributed hash tables (DHash and Pastry, respectively), where every version of the file is maintained in the distributed hash table. In these approaches, a user must ask for a file by name *and version*; the latest version can be obtained from the file's owner or from a trusted third party. This model is exemplified by CFS, which retrieves the file as follows: first, it cryptographically hashes the filename and version, receiving a key from which the file's current block map can be fetched. This map contains an ordered list of keys corresponding to the individual blocks in the requested version of the file. Each block can be fetched using its key; the key actually represents the hash of the block's content. This content-based hashing scheme for blocks has a two key benefits: (i) if two files or file versions share a page with identical content, CFS will store only one copy of the page and will use it in both files; (ii) CFS will employ caching as blocks are requested, and the cache lookup mechanism can be based on hashing, rather than naming, content.

Owner-Resolver

Coda [12] relaxes the single-owner scheme described above, in allowing data to be replicated throughout a network, and for changes to be made to the replicas. Coda's focus is on allowing updates in the presence of network partition: nodes might need to make changes without having access to the primary copy. Once connectivity is restored, the newly modified replica must be reconciled with the original data and any other changed replicas; Coda does this by sharing and replaying logs of changes made to the different replicas. If Coda determines that multiple concurrent changes were made, then activates an application-specific *conflict resolver* that attempts to resolve the conflicts. In the worst case, the data may need to be branched.

Bayou [9] uses a very similar scheme, except that changes to replicas are propagated in pairwise fashion across the network (an *epidemic protocol*) instead of sent directly to each file's owner. Nodes maintain logs of the updates they know about, including other nodes' updates; as they communicate, they exchange logs and merge them in pairwise fashion (this is an *epidemic protocol*). If conflicts occur, an application-specific *merge procedure* is triggered. Eventually the logs reach a primary node, which determines the final order of updates.

Building even further on the notion of epidemic protocols, work by Datta et al. [6] focuses on settings in which conflicts are unlikely to arise at all. It adopts an epidemic protocol that provides eventual consistency across the network. In this model, a peer that makes an update pushes a notification to subset of its neighboring peers, who may in turn forward to additional peers. This is likely to keep many of the peers "relatively" up to date. When a peer has not received an update in a while, or if it comes back online, it tries to determine the latest state by executing a "pull" request.

Consensus

Coda and Bayou allowed for concurrent updates, but relied on the holder of the primary copy to resolve conflicts. An alternative is to reconcile conflicts through some sort of voting or consensus scheme. Like Bayou, Deno [4] uses an epidemic protocol, in which nodes share information in pairwise fashion about new updates. Here, updates are grouped into transactions that are to be atomically committed or aborted. For each transaction that is not *blocked* (i.e., waiting for another transaction to complete), a distributed vote is executed to determine whether the transaction should be committed. Nodes gossip by exchanging information about transactions and votes; ordering information is maintained using version vectors [15]. If a majority of nodes vote for the transaction, rather than any other transactions that have conflicting updates, then the transaction is committed.

A number of filesystems, including BFS [3], OceanStore [13], Farsite [2], make use of *quorums* of nodes that manage the sequencing on updates: these nodes essentially serve very similarly to the single owner schemes described previously, in that they must be contacted for each update in sequence, and they define the serialization order of updates.

Partial Divergence

Two more recent works – one focused on filesystems, and the other on database instances – enable a scheme

for managing inconsistency based on peers' individual *trust policies*.

Like CFS, Ivy [14] is a filesystem built over the DHash distributed hash table. However, Ivy provides NFS-like semantics including the ability to modify files, and it does so in a novel way. Ivy has one update log per peer, and such logs are made publicly available through DHash. As a peer modifies a file, it writes these changes to its own update log (annotating them with version vectors [15] so sequencing can be tracked). As the peer *reads* a file, it may consult *all* logs; but it may also ignore some logs, depending on its local trust policy. However, Ivy assumes that it is highly undesirable for each peer to get a different version of a file (because this would prevent sharing); hence, sets of peers share a *view* of the file system – all files in the view have the same consistent version.

Finally, the Orchestra [16] *collaborative data sharing system* focuses on sharing different database instances in loose collaborations: here, multiple peers wish to share updates with one another, but each peer may selectively override the updates it receives from elsewhere. Each peer may adopt its own *trust policies* specifying (in a partial order) how much it trusts the other peers. Orchestra is specifically motivated by scientific data sharing: for instance, organizations holding proteomics and genomics data wish to import data from one another, and to refresh this imported data; however, since the data is often unreliable, each peer may wish to choose the version from the site it trusts most, and then independently curate (revise, correct, and annotate) the resulting data.

In Orchestra, as in Deno, all updates are grouped into atomic transactions. Here, each peer uses its local trust policies to choose among conflicting transactions – such a transaction will be *accepted* by the peer, and all conflicting transactions will be *rejected*. If a transaction X from a particular peer is rejected, then all subsequent transactions from that same peer that directly modified the results of X are also transitively rejected. In this trust model, the common case is that transactions come to a target peer C from a trusted peer and does not conflict with any others; hence they are immediately applied to C. Otherwise, trust composes as follows: as a transaction is propagated from peer A to peer B to peer C, it is only applied at C if it is the most trusted transaction at each step; if multiple conflicting transactions can be applied at C, then the one most trusted by C is applied. Orchestra allows instances to "partially diverge" where there are conflicts, while still maintaining sharing for portions of the data where there are no conflicts.

Key Applications

Recent applications of peer-to-peer technologies include distributed network monitoring, distributed stream processing, and exchange of data among collaborating organizations that host scientific databases. In all of these settings, data may be frequently updated.

Future Directions

One of the most promising directions of future study is the interaction between transactions and *mappings* or conversion routines: in many real-world applications, data is being shared between sites that have different data representations. There is some work (e.g., that related to view maintenance) that shows how to translate updates across mappings; but there is no scheme for mapping *transactions*.

Cross-references

- ▶ Distributed Concurrency Control
- ▶ Distributed Databases
- ▶ Epidemic Algorithms
- ▶ Peer-to-Peer Storage
- ▶ Transaction

Recommended Reading

1. Balazinska M., Balakrishnan H., and Stonebraker M. Demonstration: Load management and high availability in the Medusa distributed stream processing system. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004.
2. Bolosky W.J., Douceur J.R., Ely D., and Theimer M. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In Proc. 2000 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Comp. Syst., 2000.
3. Castro M. and Liskov B. Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst., 20(4): 2002.
4. Cetintemel U., Keleher P.J., Bhattacharjee B., and Franklin M.J. Deno: a decentralized, peer-to-peer object-replication system for weakly connected environments. IEEE Trans. Comput., 52(7):943–959, Jul 2003.
5. Dabek F., Kaashoek M.F., Karger D., Morris R., and Stoica I. Wide-area cooperative storage with CFS. In Proc. 18th ACM Symp. on Operating System Principles, 2001.
6. Datta A., Hauswirth M., and Aberer K. Updates in highly unreliable, replicated peer-to-peer systems. In Proc. 23rd Int. Conf. on Distributed Computing Systems, 2003.
7. Diao Y., Rizvi S., and Franklin M.J. Towards an Internet-Scale XML Dissemination Service. In Proc. 30th Int. Conf. on Very Large Data Bases, 2004.

8. Druschel P. and Rowstron A. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In Proc. 8th Workshop on Hot Topics in Operating Systems, 2001.
9. Edwards W.K., Mynatt E.D., Petersen K., Spreitzer M.J., Terry D.B., and Theimer M.M. Designing and implementing asynchronous collaborative applications with Bayou. In Proc. 10th Annual ACM Symp. on User Interface Software and Technology, 1997.
10. Fuxman A., Kolaitis P.G., Miller R.J., and Tan W.C. Peer data exchange. In Proc. 24th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems, 2005.
11. Huebsch R., Hellerstein J.M., Lanham N., Loo B.T., Shenker S., and Stoica I. Quering the Internet with PIER. In Proc. 29th Int. Conf. on Very Large Data Bases, 2003.
12. Kisler J. and Satyanarayanan M. Disconnected operation in the coda file system. ACM Trans. Comput. Syst., 10(1), 1992.
13. Kubiatowicz J., Bindel D., Chen Y., Czerwinski S., Eaton P., Geels D., Gummadi R., Rhea S., Weatherspoon H., Weimer W., Wells C., and Zhao B. OceanStore: an architecture for global-scale persistent storage. In Proc. 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 2000.
14. Muthitacharoen A., Morris R., Gil T.M., and Chen B. Ivy: A Read/Write Peer-to-Peer File System. In Proc. 5th USENIX Symp. on Operating System Design and Implementation, 2002.
15. Parker Jr. D.S., Popek G.J., Rudisin G., Stoughton A., Walker B.J., Walton E., Chow J.M., Edwards D.A., Kiser S., and Kline C.S. Detection of mutual inconsistency in distributed systems. IEEE Trans. Software Eng., 9(3), 1983.
16. Taylor N.E. and Ives Z.G. Reconciling while tolerating disagreement in collaborative data sharing. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2006.

Updates through Views

Yannis Velegrakis
University of Trento, Trento, Italy

Definition

Views are windows to a database. They provide access to only a portion of the data that is either of interest or related to an application. Views are relations without independent existence, as is the case of database tables created with the "create table" command. The contents of the instance of a view are determined by the result of a query on a set of database tables, typically referred to as *base tables*. Applications deal with views the same way they deal with any other relation. In fact, an application is rarely aware of whether a relation it is accessing is a base table or a view. This means that the application may issue updates on the view relation as it would have done with any other database table. Since the view instance depends on the instances of the base tables, to execute an update on the view one needs to find a number of base table modifications whose effect on the view instance is the modification described by the update command.

Historical Background

Views are one of the oldest concepts in computer science. They appeared almost at the same time with queries. In the early 1980s, as database vendors were moving towards the relational model, Codd introduced twelve rules that need to be satisfied by a database management system in order to be considered *relational*. Rule number six was referring to the concept of views and in particular, to the ability of the views to be updated when they are *theoretically updatable*, a concept that will be explained below.

The scientific literature contains different definitions (and uses) of the term "view," with two of them being dominant. The first is that a view is a relation whose instance depends (somehow) on the data of its base tables, and the second is that a view is a short-hand for a query. For query answering, those two definitions are equivalent, but for the purpose of updating, the definition that is considered makes a difference. They have different consequences. Considering the view as a query, implies that at any given point in time the instance of a view is fully specified by the results of the view query over the data, i.e., the set of possible instances of a view is the set of possible relations that can be generated as a result of the view query. A consequence of this is that the view might not be *theoretically* updatable, which means that there are updates on the view that cannot be translated to updates on the base tables. Recently, Kotidis et al. [8] have relaxed that requirement to accept views whose instance may be different from the result of their view query in order to allow arbitrary updates on the view.

Update propagation is one of the main issues in view updates. When multiple views share the same piece of data from the base tables, all these views should have consistent instances. This means that when some part of the data is updated, this change must propagate accordingly to the base tables and the other views. For the case of materialized views, and updates on the base tables, this is a well-studied problem, namely the view maintenance [10]. View maintenance has considered how to change the instances of the views in response to base table modifications.

However, updates on the views have not been studied to the same extend.

Foundations

A *database* D = <S, I> is a collection of data along with a description of its structure. The collection of data is referred to as the *database instance* I and the description of its data structures as the *database schema* S. In the relational model, a database instance is a set of relations, and the schema is the set of schemas of these relations. The contents of a relation, i.e., its instance, may be explicitly specified by a user or an application, or they may be derived from the contents of other relations. Relations of the first kind are those constructed using the "create table" command and are often referred to as *base tables*. Relations of the second kind are referred to as *views*. A view is accompanied by a query which is referred to as the *view query* and specifies how its contents will be derived from the contents of the base tables.

The term "update" on a relation implies an insertion, a deletion, or a value modification of one or more tuples. The literature often chooses to ignore modifications on the basis that a modification can be modeled as a deletion followed by an insertion. The same assumption is followed here.

Let U denote an update command on a relation R, |R| the instance of R and |U| the tuple(s) that need to be inserted in |R| (if U is an insert command) or deleted from |R| (if U is a delete command).

The *implementation* of an update U on a relation R is a new relation R′ such that:

- $|R|' = |R| - |U|$ if U is a delete command, and
- $|R|' = |U| \cup |R|$ if U is an insert command

In the case of a base table, the implementation of an update is straight forward (it is done by directly modifying the contents of the base table instance according to the update command). In the case of a view the situation is different. Since the view instance depends on the instances of its base tables an implementation is performed by translating the view update command into a series of updates on the base tables so that the requested update is observed in the view instance. Figure 1 provides a graphical representation of this notion. $Q_v(I)$ denotes the application of the view query Q_v on the database instance I. The result of this query is the instance |V| of the view V. Given an update command U on the view V, the *view update problem* is defined as the problem of finding an update W, referred to as the *translation* of U, on the instance I such that when the view query is applied on the new modified instance W(I), the new view instance $|V|' = Q_v(W(I))$ is an implementation of the update U.

For a given update U, there may be multiple possible translations. The following example illustrates such a situation. Consider a database instance that consists of a table *Personnel (Department, Employee)* with tuples {[Administration, Smith], [HR, Smith], [Research, Kole]} and the view

V_1: *select* Department *from* Personnel *where Employee* = "Smith"

The instance of view V_1 consists of the two tuples [Administration, Smith] and [HR, Smith]. Let U_1 be an update command that requests the deletion of tuple [Administration, Smith] from V_1. There are many possible changes that can be done on the instance of *Personnel* in order to make tuple [Administration, Smith] disappear from the instance of view V_1. The most obvious one is to simply delete tuple [Administration, Smith] from Personnel. Alternatively, one could update the value of its *Employee* attribute from "Smith" to *null* or to some value other than "Smith." Even changing the *Department* attribute from "Administration" to some other value "Y" will have the desired effect of removing tuple [Administration, Smith] from the view. Thus, each of the aforementioned changes constitutes an implementation of the view update U_1. They are not, however, equivalent. Each translation may be considered appropriate in different situations. The first translation, for instance, achieves the desired result and only this by deleting only one tuple from the database instance. The second achieves the result without reducing the number of tuples in the database instance, and the latter will cause the appearance of the additional tuple [Y, Smith] in the view, which makes the specific translation less appealing.

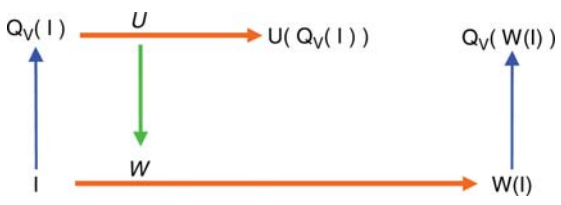
For the case of a view deletion, the first step in finding a view update translation is to find the provenance of the tuples that are to be deleted from the view, i.e., the tuples in the base tables that are responsible for the appearance in the view of the tuples that are to be deleted [4,5]. Researchers have concentrated their efforts in detecting the minimum number of changes that need to be made on the base tables in order to achieve the view update (*source-side-effect*). These changes, as mentioned previously through the

example, may result in additional changes in the view instance apart from the one requested by the view update command. Thus, another important desideratum is to find translations that minimize the number of changes in the view (*view-side-effect*). In certain cases, these two desiderata may conflict.

Finding a translation of a view deletion that minimizes the changes in the base tables or/and the view is in general an NP-hard problem, even with monotone view queries. The complexity remains NP-hard even for the very restrictive classes of views with *select-project-join* or *select-join-union* view queries [4].

A different approach to the view update problem is to deal with it at the semantic level. This approach has been advocated by many early researchers [1]. Their position is that at the moment of view definition, the data administrator needs to specify how each update on the view will be translated to updates on the base tables. This is a safe approach that guarantees a semantically correct translation. However, this approach may be hard to apply in certain practical situations. One reason is that views may have been defined at a point in which updates were not of interest and the correct translation semantics may be difficult to infer. Furthermore, even if the data designer is aware of the existence of updates on the view, she may not be able to predict at the moment of view definition all the different semantics of the possible view update commands. For instance, the deletion of the [Administration, Smith] tuple in the example above may be simply because Smith stopped working for the Administration department, or because he moved from the Administration to another department. Another reason why this approach is difficult to implement is that views may have been introduced as replacements of tables in cases of physical or logical data reorganization, in which case again, more than one alternative translation may be semantically correct.

Keller [7] has provided a detailed classification of the different kinds of view update translations for projections, selections, select-project-join, and select-project-join-union views. Through an interactive procedure with the data administrator at the time of the view definition, the system tries to infer the right translation rules. At the other end, Dayal and Bernstein [6] have studied the classes of views for which the translation of an update is not ambiguous. For the case of translation ambiguity or the case in which it is possible to implement the update but with more changes in the view instance than those the view command requested, then the update may not be allowed.



Updates through Views. Figure 1. The updates through views problem.

A different approach in finding a view update translation is to reduce it to a constraint satisfaction problem [9]. Relational views can be represented as conditional tables and a view update can be translated into a disjunction of a number of constraint satisfaction problems (CSPs). Solutions to the CSPs correspond to possible translations. For the case of multiple candidate translations, semantic information to resolve ambiguity can be modeled as additional constraints.

The problem of updates through views appears also in other models, such as the object views or XML. Since most of the studies have been done for the relational model, one way to deal with the problem of updating object or XML views is to abstract it to the relational case [2,3].

Key Applications

Information Integration Systems. IIS are used to provide a unified view of data that is stored in multiple heterogeneous and physically distributed data sources. The majority of such systems are read-only, since data maintenance is considered to be an exclusive responsibility of the owner of each data source. This situation is gradually changing. Integrated views provide an excellent opportunity for detecting errors and inconsistencies among the different data sources. Once these inconsistencies are detected and corrected at the view level, the data in the sources will have to be accordingly updated.

Corporate Environments. Multiple different systems may operate in different parts of an organization, but centralized data management is crucial for decision making and policy implementations. Views provide the mean to propagate changes from the centralized level to the individual systems of the organization.

P2P. P2P systems use views to describe the relationships between the contents of one peer and its

acquaintances. When a modification takes place in one of the peers, the views are used to determine its effects (if any) on the contents of its adjacent peers.

Database Management Systems. View support has been prevalent in Database Management Systems. They provide physical data independence, i.e., decoupling the logical schema from the physical design which is usually driven by performance reasons. By controlling access to the views, one can control access to different parts of the data and implement different security policies. It is not uncommon the case of databases that provide access to their data exclusively through views. For them the ability to perform arbitrary updates on the views is becoming a necessity since the only way to update the stored information is by issuing updates on the views.

WEB. The World Wide Web has evolved to be the world's largest database where organizations and individuals publish their information and data. Web users are then using applications and integration engines to query and retrieve that information. Since the users have no access to the individual sources, the web in its current form is a read-only system, restricting the potential of its million users. If the Web is to be brought from its read-only status into a full-fledged database, allowing updates on the web views to propagate to the sources is a fundamental requirement.

Cross-references

► Provenance
► Side-Effect-Free View Updates
► View Maintenance

Recommended Reading

1. Aaron B., Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: a language for updatable views. In Proc. 27th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2006, pp. 338–347.
2. Barsalou T., Siambela N., Keller A.M., and Wiederhold G. Updating relational databases through object-based views. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 248–257.
3. Braganholo V.P., Davidson S.B., and Heuser C.A. On the updatability of XML views over relational databases. In Proc. 6th Int. Workshop on the World Wide Web and Databases, 2003, pp. 31–36.
4. Buneman P., Khanna S., and Tan W.C. On propagation of deletions and annotations through views. In Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2002, pp. 150–158.
5. Cui Y., Widom J., and Wiener J.L. Tracing the lineage of view data in a data warehousing environment. ACM Trans. Database Syst., 25(2):179–227, 2000.
6. Dayal U. and Bernstein P. On the correct translation of update operations on relational views. ACM Trans. Database Syst., 8(3):381–416, 1982.
7. Keller A.M. Choosing a view update translator by dialog at view definition time. In Proc. 12th Int. Conf. on Very Large Data Bases, 1986, pp. 467–474.
8. Kotidis Y., Srivastava D., and Velegrakis Y. Updates through views: a new hope. In Proc. 21st Int. Conf. on Dara Engineering, 2005.
9. Shu H. Using constraint satisfaction for view update translation. In Proc. 21st Int. Conf. on Data Engineering, 2005.
10. Widom J. Research problems in data warehousing. In Proc. Int. Conf. on Information and Knowledge Management, 1995, pp. 25–30.

URI

► Resource Identifier

Usability

Nigel Bevan
Professional Usability Services, London, UK

Synonyms

User centered design

Definition

A product is usable if the intended users can achieve their goals with effectiveness, efficiency and satisfaction in a specified context of use [8]. Usability is achieved by taking a user-centered approach to design, and thus ensuring that the product incorporates characteristics that support usability.

Historical Background

Usability was adopted as a technical term to replace the phrase "user friendly," which by the early 1980s had acquired undesirably vague and subjective connotations. It is a goal for product design in the scientific fields of HCI, human factors and ergonomics. In 1985

Gould and Lewis [5] described the central principles of what became known as user-centered design [15]: (i) early focus on users and tasks., (ii) empirical measurement, (iii) iterative design.

Usability has since grown into an established discipline with the Usability Professionals Association founded in 1991, and the landmark book by Nielsen on Usability Engineering [13] published in 1993.

Although the field of usability overlaps with human factors and ergonomics, it tends to be associated with systems that have discretionary users. Human factors and ergonomics have traditionally focused on how human capabilities can be integrated into pre-defined tasks and work systems. Usability is more typically concerned with designing applications to support users and tasks that are often not well-understood.

Foundations

The broad interpretation of usability is the user's experience of the quality of the system in use [1]: to what extent the user is successful in achieving their goals (effectiveness), in an acceptable amount of time (efficiency), without undesirable side-effects (safety), and in a way that satisfies the user? While this interpretation is preferred by people working in the field, in systems development usability is often interpreted more narrowly as ease of use (as in the 1991 definition of usability in ISO 9126 [10]). More recently it has been appreciated that the prerequisites for achieving usability as a goal include not only software and hardware with appropriate functionality, reliability and ease of use, but also usable data [12].

The broad interpretation of usability is thus a black box view at the system level, measured in terms of success in achieving goals for effectiveness, efficiency, safety and satisfaction (Fig. 1). It is a result of the user's interaction with the product, and is facilitated by appropriate product characteristics.

In the context of databases, the major product characteristics for usability are software and data.

Software Usability

There are many well-established design principles and standards, both for general software usability, and for web usability. At a high level they include heuristics such as [13]:

- Use simple and natural dialogue, and speak the users' language.
- Minimize user memory load.
- Be consistent in screen and task design.
- Provide feedback in response to user input.
- Provide shortcuts and clearly marked exits.
- Prevent errors and provide informative error messages, help and documentation.

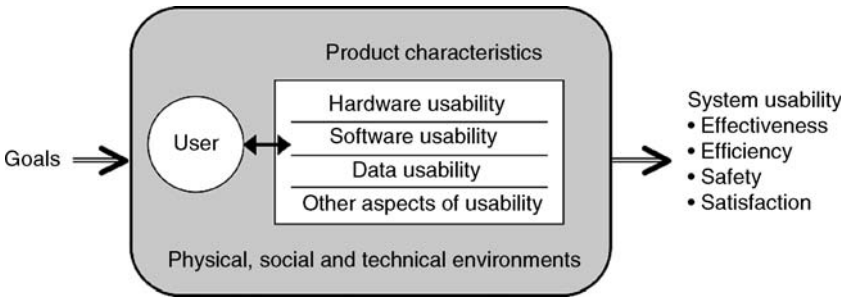
At a detailed level ISO 9241 [8] contains comprehensive interface guidelines, and the US Government has produced an authoritative and well-documented set of web design guidelines [18].

Data Usability

ISO is developing a model for data quality [7]. Data characteristics that contribute to overall system usability include completeness, depth of information, accuracy and understandability. Poor data quality makes it very difficult for users to achieve their goals, even if the user interface is easy to use.

For example, an online train timetable will have poor usability if the data is not usable, for example if:

- The destinations or types of ticket are *incomplete*.
- There is insufficient *depth of information* about the destinations or types of ticket



Usability. Figure 1. System usability.

- Any of the train times or ticket information are *inaccurate*
- The user cannot *understand* the names used for the destinations or types of ticket.

Key Applications

User Centered Design

A product will only be usable if the design and development process is based on an in depth understanding of the range and types of intended users, their task, and the physical, technical and social environments of use. Potential design solutions need to be continually evaluated from a user perspective from the earliest stages of design. The activities necessary to achieve user centered design are described at a high level in ISO 13407 [9], and in more detail in ISO 18529 [9].

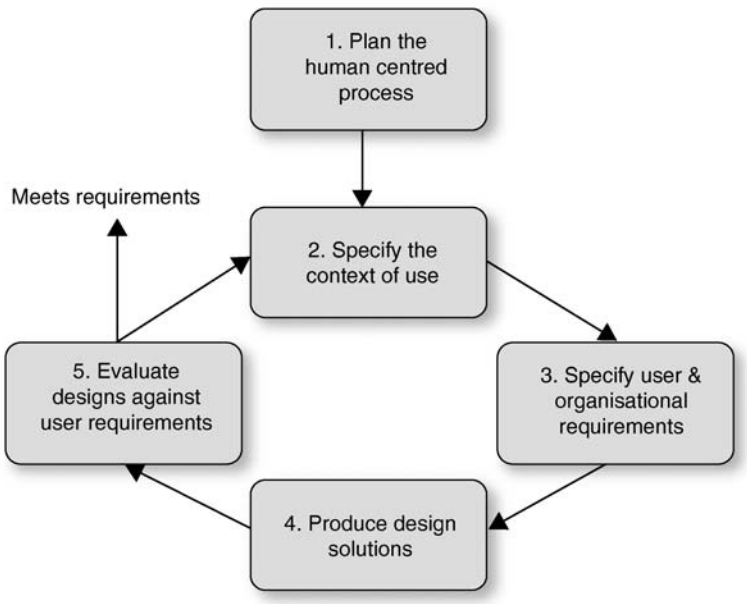
Key activities are illustrated in Fig. 2:

1. Planning. The nature and complexity of the user-centered activities should depend on the importance of usability, and a judgement on which particular activities are necessary for project success.
2. Context of Use. Detailed information about the context of use is an essential input to requirements, and provides a basis for prioritizing testing.
3. User and Organizational Requirements. These should be specified both at a high level in terms of user performance and satisfaction, and at a more detailed level of user interface characteristics.
4. Early design solutions should be produced as mock-ups, typically paper-based, simulations for exploratory testing [17]. Later in development, computer simulations can be used evaluate the user interface.
5. Designs should be evaluated from a user perspective using both expert and user-based methods, to obtain design feedback, and to establish whether requirements have been met.

Context of Use

The characteristics of the users, tasks and the organizational and physical environment define the context in which the system is used. It is important to gather and analyze information on the current context in order to understand and then specify the context that will apply in the future system. Analysis of existing or similar systems can provide information on a whole range of context issues including needs, problems and constraints that might otherwise be overlooked.

Methods for identifying the context range from collecting information in a workshop attended by the relevant stakeholders, to detailed field studies and task analysis.



Usability. Figure 2. User centered design process.

User and Organizational Requirements

User needs should be identified in conjunction with identifying the context of use. The system requirements should include criteria for user performance, for minimization of adverse effects and for user satisfaction. These are most easily set in relation to baseline values for a comparable existing system. More detailed requirements should be developed iteratively incorporating feedback from early design solutions.

Design Solutions

Using simulations, mock-ups or prototype allows designers to communicate in a meaningful way what the proposed design is/would be like to users and other stakeholders. Paper prototypes can be very effective ingathering feedback [17]. The design should be consistent with established principles for software and data usability [8,12].

Usability Evaluation

Usability evaluation will reveal the strengths and weakness in the design solution and can indicate where the design should be improved.

Feedback from usability evaluation is particularly important because developers seldom have an intimate understanding of the user's perspective and work practices. Initial designs therefore very rarely fully meet user requirements. The cost of rectifying any divergence between the design and user needs increases rapidly as development proceeds, which means that user feedback should be obtained as early as possible. Without proper usability evaluation, a project runs a high risk of expensive rework to adapt a system to actual user needs.

Evaluation of overall system usability and of detailed product characteristics are complementary. Although a user-based evaluation is the ultimate test of usability, it is not usually practical to evaluate all permutations of user types, tasks and operational conditions. Evaluation of the detailed characteristics of the product or interactive system can anticipate and explain potential usability problems, and can be carried out before there is a working system. However evaluation of detailed characteristics alone can never be sufficient, as this does not provide enough information to accurately predict the eventual user behavior in every context.

The purpose of the evaluation may be primarily to obtain design feedback to identify and fix any obstacles to effective usability, or to validate the usability of a system. The most useful validation data are measures of user performance and satisfaction obtained from measuring system usability.

When to Use Methods for Design Feedback The most common type of usability evaluation is to improve a product by identifying and fixing usability problems. Evaluation of early mock-ups can also be used to obtain a better understanding of user needs and to refine requirements. An iterative process of repeated evaluation of prototypes can be used to monitor how closely the prototype designs match user needs. The feedback can be used to improve the design for further testing. Early evaluation reduces the risk of expensive rework. Usability evaluation is most effective when it involves a combination of expert and user-based methods.

When to Use Validation Methods In a more mature design process evaluation to obtain design feedback should be complemented by establishing usability requirements and testing whether these have been achieved by using a more formal method to validate usability. This reduces the risk of delivering a product that fails as a result of poor user performance.

Similar testing of an existing system can be used to provide baseline measures that can form the basis for usability requirements (i.e., objectives for human performance and user satisfaction ratings). A Common Industry Specification for Usability Requirements [14] has been developed to support iterative development and sharing of such requirements.

Validation tests at the end of development should have acceptance criteria derived from the usability requirements. Validation methods can be elaborated to also identify usability problems, but if prior iterative rounds of usability testing are performed, then typically there will be few usability surprises uncovered during this late stage testing.

Validity Usability testing should be carried out carefully and systematically, otherwise the resulting data may not be valid or reliable. O'Hara et al. [16] describe the need for the following types of validity:

- External validity: extent to which the context of use for the test is realistic.
- Construct validity: extent to which the measures are representative of user performance and satisfaction.

- Internal validity: extent to which the test is properly designed.
- Statistical validity: extent to which the statistical conclusions are valid.

Poor usability data may lead to poor design decisions and ultimately an error-prone and unsafe product. Therefore, usability testing protocols should be developed in collaboration with professional usability specialists. Non-specialists under the direction and training of professionals can handle the actual execution and reporting of the testing.

Cost Benefits

The objective of user centered design is to ensure that products can be used by real people to achieve their tasks in the real world. This requires not only easy-to-use interfaces, but also the appropriate functionality and support for real business activities and work flows. According to IBM [6], developing easy-to-use products "makes business effective. It makes business efficient. It makes business sense."

User centered design can reduce development and support costs, increase utilization, and reduce staff costs for employers [2]:

- Development costs can be reduced by fixing usability problems early in the development process, avoiding unnecessary functionality and minimizing documentation.
- Support costs can be reduced by minimizing help line, maintenance and training costs.
- Utilization of the system by individuals and organizations can be increased as a result of higher user success rates, productivity and satisfaction.

Usability of Digital Libraries

Digital libraries pose some particular challenges for usability, as a result of the large amount of information they contain, the difficult of determining what users want, and the need to cater for browsing and searching. Digital libraries demand more sophistication of query formulation than web search engines. Skills acquired in one library environment are often not easily transferred to another [3].

Cross-references

- ▶ Data Quality Dimensions
- ▶ Human-Computer Interaction
- ▶ Information Quality Assessment

Recommended Reading

1. Bevan N. Quality in use: meeting user needs for quality. J. Syst. Softw., 49(1):89–96, 1999.
2. Bevan N. Cost benefits framework and case studies. In Cost-Justifying Usability: An Update for the Internet Age, R.G. Bias, D.J. Mayhew (eds.). Morgan Kaufmann, San Francisco, CA, 2005.
3. Blandford A. and Buchanan G. Usability of digital libraries: a source of creative tensions with technical developments. IEEE-TCDL, Bulletin, 2003.
4. Dumas J.S. and Redish J.C. A Practical Guide to Usability Testing. Ablex Publishing Corporation, Norwood, NJ, 1993.
5. Gould J.D. and Lewis C. Designing for usability: key principles and what designers think. Commun. ACM, 28(3):300–311, 1985.
6. IBM. Cost Justifying Ease of Use, 2000. Available at: www-03.ibm.com/easy/page/23
7. ISO 13407. User centred design process for interactive systems. ISO, 1998.
8. ISO 9241. Ergonomic requirements for office work with visual display terminals (VDT)s (Pts 10–17). ISO, 1997–99.
9. ISO TR 18529. Human-centred lifecycle process descriptions. ISO, 2000.
10. ISO/IEC 9126. Software product evaluation – Quality characteristics and guidelines for their use. ISO, 1991.
11. ISO/IEC CD 25010. Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality model. ISO, 2008.
12. ISO/IEC FCD 25012. Software engineering – Software product quality requirements and evaluation (SQuaRE) – Data quality model. ISO, 2008.
13. Nielsen J. Usability Engineering. Academic Press, San Diego, CA, 1993.
14. NIST. Industry Usability Reporting, 2007. Available at: www.nist.gov/iusr
15. Norman D.A. and Draper S.W. User Centered System Design New Perspectives on Human–Computer Interaction. Lawrence Erlbaum Associates, Mahwah, NJ, 1986.
16. O'Hara J., Stubler W., Higgins J., and Brown W. Integrated System Validation: Methodology and Review Criteria (NUREG/CR-6393), 1995. Available at: www.bnl.gov/humanfactors/Publications.asp
17. Snyder C. Paper Prototyping the Fast and Easy Way to Define and Refine User Interfaces. Morgan Kaufmann, San Francisco, CA, 2003.
18. U.S. Department of Health and Human Sciences. Research-Based Web Design & Usability Guidelines, 2006. Available at: www.usability.gov/guidelines

User Centered Design

▶ Usability

User Classifications

► Lightweight Ontologies

User-Centred Design

► Human-Computer Interaction

User-Defined Time

Christian S. Jensen[1], Richard T. Snodgrass[2]
[1]Aalborg University, Aalborg, Denmark
[2]University of Arizona, Tucson, AZ, USA

Definition

The concept of *user-defined time* denotes time-valued attributes of database items with which the data model and query language associate no special semantics. Such attributes may have as their domains all domains that reference time, e.g., date and time. The domains may be instant-, period-, and interval-valued. Example user-defined time attributes include "birthday," "hiring date," and "contract duration." Thus, user-defined time attributes are parallel to attributes that record, e.g., salary, using domain "money," and publication count, using domain "integer." User-defined time attributes contrast transaction-time and valid-time attributes, which carry special semantics.

Key Points

The valid time and transaction time attributes of a database item are "about" the other attributes of the database item. The valid time records when the information recorded by the attributes is true in the modeled reality, and the transaction time captures when the data item was part of the current database state. In contrast, user-defined time attributes are simply "other" attributes that may be used for the capture of information with which valid time can be associated.

Conventional database management systems generally support a time and/or date attribute domain. The SQL2 standard has explicit support for user-defined time in its `datetime` and `interval` types.

Cross-references

► Temporal Database
► Transaction Time
► Valid Time

Recommended Reading

1. Jensen C.S. and Dyreson C.E. (eds.). A consensus glossary of temporal database concepts – February 1998 version. In Temporal Databases: Research and Practice, O. Etzion, S. Jajodia, S. Sripada (eds.). LNCS, vol. 1399, Springer-Verlag, Berlin, 1998, pp. 367–405.
2. Snodgrass R.T. and Ahn I. A taxonomy of time in databases. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1985, pp. 236–246.
3. Snodgrass R.T. and Ahn I. Temporal databases. IEEE Comput., 19(9):35–42, September 1986.

User-Level Parallelism

► Inter-query parallelism

Using Efficient Database Technology (DB) for Effective Information Retrieval (IR) of Semi-Structured Text

► Integrated DB&IR Semi-Structured Text Retrieval

Utility Computing

► Storage Grid

UUID

► Resource Identifier