Software Systems Analysis— A Research Area Overview

Reema Al-Kamha

Abstract

Software systems analysis is a field in which analysts continually learn new techniques and approaches to properly capture, maintain, understand, and develop more efficient and effective software systems. We begin this research area overview by defining systems, systems analysis, and modeling. In subsequent sections, we focus on data and behavior representation of the system under study, prototyping, and formalism. Finally, we introduce some of the current work such as form-oriented analysis, fisheye views to support system analysis, and extreme programming and consider future work on software systems analysis such as extreme non-programming and new challenges for conceptual modeling.

1 Introduction

By way of introduction, it is important to define software systems, systems analysis, and modeling.

1.1 Systems

Several authors have defined a software system: "a system is an assemblage of parts forming a complex or unitary whole that serves a useful purpose" [BF90], "a system is a group of interacting objects" [EKW92], and "a system is a set or arrangement of elements that are organized to accomplish some predefined goal by processing information" [Pre01].

From these definitions we draw our definition of a software system as *a set of interacting software components that serve a useful purpose.* The components themselves may be considered to be software systems; with respect to large systems they are considered to be subsystems.

1.2 Systems Analysis

Systems analysis is the study of a system under consideration (which may be real or imagined) [EKW92]. Its purpose is *understanding* and *documentation* of the essential charac-

teristics of the system being studied. Its eventual goal is to come up with a specification of the system under study.

1.3 Modeling

A model provides the blueprints of a system [BRJ99]. A good model includes all the effective components in the system under study and ignores all elements that are not relevant. Modeling provides a better understanding and visualization of the system under study. Every system can be described from different aspects using different models. A model may emphasize components and their relationship to other components in the system under study, the behavior of each component in the system or the interactions among the components in the system under study.

2 Representation—Data

During analysis, analysts represent the data of the system under study in terms of metadata concepts and relationships. Metadata helps the analysts focus more easily on matching the data representation to the requirements of the system under study.

We consider three representations of data that are commonly used in systems analysis: conceptual data modeling, knowledge representation, and ontologies.

2.1 Conceptual Data Modeling

In software systems analysis, conceptual data models have proven to be quite successful for representing data at a higher level of abstraction. Conceptual models represent components and their relationship to other components in the system under study in a graphical way at a conceptual level. Some of the well known conceptual models are ER [Che76], ORM [Hal01], ORM [EKW92], and UML [BRJ99]. (Note that "ORM" in [Hal01] is Object-Role Modeling and is pronounced "orm", while "ORM" in [EKW92] is Object-Relationship Model and is pronounced "O"-"R"-"M".)

- Entity Relationship Model [Che76]: ER models were introduced by Chen in his classic 1976 article to represent the conceptual structure of data in a database system. The ER modeling constructs are entities, attributes, and relationships among the entities. In ER diagrams (ERD's) entities are represented as rectangles, relationships among entities are represented as diamonds connected by lines to entities, and attributes are represented by ovals hanging off lines attached to entities. Later modifications to classic ERD's include ISA and CONTAINED-IN relationships [TYF86].

- Object-Role Modeling [Hal01]: ORM consists of a set of objects (entities or values) that play roles (parts in relationships). In ORM models there are no attributes which is the main feature that distinguishes them from ER models. The number of roles in each relationship set indicates the type of the relationship set (a unary relationship set has one role, a binary relationship has two roles, a ternary relationship set has three roles). ORM has mandatory constraints, uniqueness constraints, and subset constraints. Mandatory constraints indicate that the role that an object plays is mandatory. Uniqueness constraints indicate that the object plays the role only once. Subset constraints indicate that an object is a subset of another object.

- Object-Relationship Model [EKW92]: ORM is a conceptual model consisting of object sets, relationship sets, and constraints over these object and relationship sets. An object set may be a subset of another object set in a generalization/specialization hierarchy (ISA hierarchy). An ISA hierarchy may be constrained by partition ($\uplus$), union ($\cup$), or mutual exclusion ($+$) among specializations. Any object-set/relationship-set connection may have a role, but a role is simply a shorthand for an object set that denotes the subset consisting of the objects that actually participate in the connection. An ORM also has aggregation, association relationship sets to model respectively, subpart/superpart and set/member relationships.

- Unified Modeling Language [BRJ99]: UML is an object-oriented modeling approach. UML data diagrams consist of classes (rectangles) and relationships that connect classes (paths with different kinds of lines used to distinguish the kinds of relationships). Classes represent the objects in the system under study. Classes must have a name, and may have attributes and operations. The relationships include dependency, generalization, association, role, or aggregation relationships.

2.2 Knowledge Representation

The objectives of knowledge representation are: (1) having a representation that is rich enough to represent all needed information, (2) having a representation that is as close to the system under study as possible, and (3) having a representation that is as robust as possible, so that small changes in the system under study result in small changes in the representation. Various techniques for knowledge representation are used in systems analysis, but semantic networks [Gri82] and frames [Min75] are the most common.

- Semantic networks [Gri82] are the most popular graphical method for representing knowledge. This technique uses object-oriented concepts to represent knowledge,

making use of classes, class properties, object instances, and inheritance. Semantic networks are used basically as a visual representation. Common relationships are is-a, has-a, owns, and made from.

- Frames [Min75] have proven to be good models for representing real-world objects. A frame is a holistic data structure based on object-oriented programming technology. A frame contains related knowledge about an object, which is derived from a class to which the object belongs. Frames can show complex relationships, graphic information, and inheritance in a concise manner. A frame is a collection of slots and fillers that define an object. A slot is like a field within a table that holds values, called fillers, within a frame.

2.3 Ontologies

An ontology in philosophy refers to a conceptualization of what can exist or what can be in the world [Bun77]. Ontologies have been used as a source of theory to investigate tools and techniques used in the analysis and design of information systems. A key development in the use of ontologies for the study of information systems is the work of Wand and Weber [WW90], based on Bunge's ontology [Bun77, Bun79]. A common working definition of an ontology is Gruber's statement that an ontology is *an explicit specification of a shared conceptualization* [Gru93].

Ontologies are used in systems analysis for the following three reasons: (1) ontologies facilitate the process of identifying the requirements of the system and understanding the relationships among the components of the system under study; (2) ontologies improve the reliability of software systems; and (3) ontologies facilitate the design of reusable systems.

Many concrete ontologies have been developed. In knowledge representation, well known contributions include Ontolingua [Gru93], CYC [LG90], and the XML based schemas, the latest of which is OWL [HPSH03].

- Ontolingua [Gru93]: The Ontolingua language is based on KIF (Knowledge Interchange Format) and the Frame Ontology. KIF has a declarative semantics and is based on first-order predicate calculus. It provides definitions for object, function, relation, and logical constants. KIF is a language for knowledge exchange and is tedious to use for the development of ontologies. Thus, the Frame Ontology is built on top of KIF and provides definitions for object-oriented and frame-language terms like class, subclass-of, and instance-of. Ontolingua lets the developer decide whether to use the full expressiveness of KIF, where axioms can be expressed, or to be more restrictive during the specification by using only Frame Ontology terms. An ontology

developed with Ontolingua is typically defined by: relations, classes (treated as unary relations), functions (defined like a relation), individuals (distinguished objects), and axioms (relating these terms).

- CYC [LG90]: CYC provides a foundation for common sense reasoning by developing ontologies for a wide variety of domain-specific applications. Knowledge in CYC is represented in the form of assertions in a variant of first-order-logic called CYCL. The CYC knowledge base itself contains simple assertions, interface rules, and control rules for its interface; an interface engine can be used to drive new assertions using this knowledge base.

- Web Ontology Language (OWL) [HPSH03]: OWL is a standard ontology language for the web and an interchange format for ontologies. It unifies three useful aspects for ontologies: knowledge modeling primitives provided by frame systems; formal semantics and efficient reasoning support from description logics; and a syntax compatible with the Web standard. It provides three sub-languages that meet different users' needs: (1) OWL-Lite can provide a class hierarchy together with some simple constraints, (2) OWL-DL is based on description logic formalisms which allow for powerful logic expressions, and (3) OWL Full enables maximum expressiveness, without regard for computational limits. OWL is an extension of RDF and RDF Schema.

There is no strict line between ontologies and conceptual data models, but ontologies are typically more general and more reusable; are intended for multiple purposes, goals, and users; are more easily shareable; and take a stronger stand on semantics of concepts.

3 Representation—Behavior

An important goal of analysis is to capture and communicate not only the static aspects of a system under study, but also its dynamic behavior. Behavior may be the behavior of each component in the system or interactions among the components in the system under study. There are several behavior models commonly used to describe the behavior of the system under study including Petri nets [Pet77], finite-state machines [Cho78], statecharts [Har87], and state nets [EKW92].

- Petri Nets (PNs): Petri nets were first introduced in 1962 by C.A. Petri and were described well by Peterson [Pet77]. A Petri net model has two types of nodes: places and transitions.

Places can contain tokens which are used to simulate the dynamic and concurrent activities of systems. Once tokens are assigned to places, the resulting Petri net is said to be marked. A transition is enabled if each of its input places contains at least one token. Only enabled transitions can fire, and when a transition fires it consumes a token from each input place and produces a token for each output place.

There are three kinds of arcs in petri net graphs: input arcs, output arcs, and inhibitor arcs. Input arcs are arrow-headed arcs from places to transitions; output arcs are arrow-headed arcs from transitions to places, and inhibitor arcs are circle-headed arcs from places to transitions. In Petri-net modeling, places can represent conditions, and transitions can represent events. A token arriving at a place can be interpreted as a true condition.

Petri nets have gained popularity in recent years because of their usefulness in modeling and analyzing concurrent systems. However, the concept of time is not explicitly provided in Petri nets, which limits their usefulness for real-time systems.

- Finite State Machines (FSM) [Cho78]: Finite state machines (FSM) are used in many reactive systems to describe the dynamic behavior of an object based on its state. Early on [Cho78] suggested using finite-state models to design and test small software components. Finite state machines can be represented as graphs where states are represented by circles and arcs are labeled with an event and an (optional) output event. A transition occurs if the system is in the state at the beginning of the arrow and the event on the transition occurs. The action is executed, and afterwards the system is in the state at the end of the arrow. Finite state machines cannot model concurrent systems.

- Statecharts [Har87]: Statecharts extend traditional FSMs to include hierarchy, synchronized concurrency, and global communication. Statecharts consist of states, transitions, and actions. A transition has a trigger. The triggers of the transitions are all events of two types: external events that come from external sources and internal events that come from internal sources. The trigger of a transition may also include a condition. A transition can be labeled not only with a trigger that causes it to be taken, but also, optionally with an action separated from the trigger by a slash. When the transition is taken, the specified action is carried out instantaneously.

- State Nets [EKW92]: State nets are used to model the behavior of object instances of an object class. Every object set has a state net that documents states, transition conditions, and actions for that object set. The state net consists of states

and transitions. Transitions are represented as rectangular boxes with a horizontal dividing line. The upper part of a transition box contains its trigger. A trigger gives the condition that, when met, may cause the transition to fire. The lower part of a transition box contains its action, if any. Actions are operations that take place when the transition fires. Arrows (directed arcs) connect one or more states to a single transition. Arrows may have single-heads/single-tails, single-heads/multiple-tails, or multiple-heads/single-tails. Arrows that point to transitions are called in-arrows, while arrows that point away from transitions are called out-arrows. An in-arrow always has a single head. An out-arrow always has a single tail. State nets can have both inter-object concurrency and intra-object concurrency. Inter-object concurrency happens when several objects are in various states and transitions at the same time. Intra-object concurrency happens when an object instance is in more than one state or transition of a single copy of a state at any point in time.

4 Prototyping

Prototyping is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem [Dav92]. It is a partial implementation because if it were a full implementation, it would be the system, not a prototype of it.

There are at least two types of prototyping—throwaway and evolutionary.

- In the throwaway approach, the prototype software is constructed in order to learn about the problem or its solution and is usually discarded after the desired knowledge is gained.

- In the evolutionary approach, the prototype is constructed in order to learn more about the problem or its solution. Once the prototype has been used and the required knowledge gained, the prototype is then adapted to satisfy the new better understood needs. The prototype is then used again, more is learned, and the prototype is re-adapted. This process repeats indefinitely until the prototype system satisfies all needs and has thus evolved into the real system.

Exemplary descriptions of prototyping include [BBC91, Hab91, JEW95].

5 Formalism

Formal methods are mathematically based techniques for describing system properties. Formalization can lead to models that are: (1) consistent such that no contradiction remains

between specified components; (2) complete which guarantees all the needed information is present; and (3) unambiguous such that all information used is precisely defined.

At present, formal methods have not gained the level of use that many had hoped for. There are several reasons: (1) formal specification focuses on function and data (timing, control, and behavioral aspects of a problem are more difficult to represent); (2) some elements of a problem (e.g. human/machine interfaces) are better specified using graphical techniques or prototypes; and (3) formal methods typically require understanding of various types of logics, set theory notation, and predicate calculus. Although formal methods are not as yet used widely in the industry, when used they do offer substantial advantages over less formal techniques.

A variety of formal specification languages are in use today, including Communicating Sequential Processes (CSP) [Hoa85], Vienna Development Method (VDM) [Jon91], and Z [Spi89].

- Communicating Sequential Processes (CSP) [Hoa85]: CSP is a formal language used to describe parallel systems. CSP describes processes in terms of entities (processes) which interact using events (which can be thought of as messages). The representation has two halves: processes can be described in terms of the events in which they may participate, and groups of processes can be described in terms of the traces of events in which they participate.

- The Vienna Development Method (VDM) [Jon91]: VDM is based on set theory and first order predicate calculus. A VDM specification focuses on the functions of the system that define 'what' the system does. The result of a VDM design process is a series of specifications in which each succeeding specification is less abstract and closer to the actual implementation than the preceding specification. Each specification is tied to the preceding specification by a correctness argument. Each VDM specification, regardless of its level of abstraction, can be viewed as a set of state descriptions. Each state description is made up of a set of state variables, operations on the state variables, and invariants on the state variables defined as pre and post-conditions. These invariants form the correctness argument for the specification.

- Z [Spi89]: Z is based on typed set theory and first order predicate logic. Z specifications are structured as a set of schemas that introduce variables and specify the relationships between these variables.

Another example of formalism is Description Logics (DLs) [BCM$^+$03] which are formalisms commonly used for knowledge representation. The semantic characterization of

DLs is based on first-order logic. DLs represent the knowledge of an application domain first by defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain. The basic building blocks of DLs are concepts, roles, and individuals. Concepts describe the common properties of a collection of individuals and can be considered as unary predicates which are interpreted as sets of objects. Roles are interpreted as binary relations between objects. Description logic systems have been used for a variety of applications including conceptual modeling, software management systems, planning systems, and configuration systems.

6 Recent Work

Some of the current research related to software systems analysis are form-oriented analysis, fisheye views to support systems analysis, and extreme programming.

6.1 Form-Oriented Analysis

Form-Oriented Analysis [DW04] is a new analysis technique for form-based applications. Form-based style systems present to the user at each point in time a page that offers information as well as a number of interaction options, typically forms. If the user has filled out a form and submits the form, the system processes the data and generates a response page. This response page again offers different interaction options to the user. A complete system specification is given in four documents: a form chart, a dialogue constraint annotation, a data dictionary, and semantic data model.

A form chart models the system interface as a bipartite state transition diagram that alternates between two kinds of states. The first kind of state is called a client page, which represents the form that a user fills. The system remains in such a client page state until the user triggers a page change. The second kind of state is called a server action. Server actions represent the system's action in response to a page change. These states are left automatically by the system and lead to a new client page. Each change in a client page leads to a server page. The transitions leaving a client page are called page/server transitions or options; the transitions from server actions to pages are called server/page transitions. The form chart is annotated by declarative dialogue constraints, written in Dialogue Constraint Language (DCL), an extension of Object Constraint Language (OCL). The data dictionary represents each client page and server page as a rectangle that has a name of the page (client page name or server page name) and contains the information shown on the page with their types. The semantic data model represents the system state between user interactions.

6.2 Using Fisheye Views to Support Systems Analysis

A recent technique introduced in [TSSO04] uses fisheye views as an aid to systems analysis and design. The idea is to make the conceptual models able to represent an entire system and a specific part of the system at the same time. The designer in this case can see a subprocess in the context of the entire system, while emphasizing a subprocesses of interest by enlarging the scale. The advantages of using fisheye views are to increase the effectiveness of system design due to the ability to recognize and eliminate redundancy and to effectively see linkages between subsystems.

Actually, this is similar to high-level abstract views introduced more than decade ago in [EKW92]. High level abstractions reduce the complexity in large models. Their purpose is to represent fundamental system concepts, whereas lower level abstractions unfold supporting detail. Rather than using fisheye views, high-level constructs are shaded. When they are imploded, the shading shows that the designer can explode them to reveal more information. When they are exploded, the shading helps show the extent of the high-level construct.

6.3 Extreme Programming (XP)

Although some people consider XP primarily as programming, it also includes the understanding and documentation steps of systems analysis. XP was originated by Kent Beck [Bec00] as a new methodology for small-sized teams developing software with vague or rapidly changing requirements. XP techniques can be viewed as methods for rapidly building and disseminating institutional knowledge among members of a development team. The goal is to give all developers a shared view of the system which matches the view held by the users of the system. To this end, XP favors simple designs, metaphors, collaboration of users and programmers, frequent verbal communication, and feedback. In XP, the analyst and the developer are no longer separate; they are basically united in a single unit. In XP, progress during software development is measured and tracked based on the how well the code implements the observable behavior of the system. The code is the model. XP discourages documentation by making the requirement from the team only the code.

When developing software systems with XP, the customer is asked to sit with the XP development team. Early on, the customer describes the system as a set of stories which are high-level statements of what the features of the system to be implemented should be. Stories are recorded on index cards. Developers estimate how long each story will take to implement. The customer decides, based on value and cost, the order in which stories will be developed. The team works iteratively: the customer writes tests and answers questions

while the programmers program. The XP customer has frequent opportunities to change the team's direction if circumstances change. Because testing is so prominent, the customer is aware of the project's true status much earlier.

Some of the drawbacks of XP are: (1) it is designed for a single small team of fewer than a dozen team members; therefore it has problems scaling up for large projects; and (2) XP may not be applied to a system where planning ahead is necessary.

7 Summary and Future Directions

In conclusion, software systems analysis has been here for a long time, and it covers a lot of things. We introduced some of its major aspects such as data and behavior representation of the system under study, prototyping, and formalism. In addition, we presented some of the recent work in that field. As we conclude, we mention some of the future directions in the field: extreme non-programming and new challenges for conceptual modeling.

7.1 Extreme Non-Programming (XNP)

Tony Morgan in his keynote address at ISTA in July 2004, introduced the idea of extreme non-programming, which is a challenge in developing software systems. He motivated the change in the development process of software systems as follows: (1) the overall process of developing software has not changed much since the 1960s; (2) currently produced software is expensive, takes a long time to develop, and has low quality; and (3) most problems arise during analysis and specification in software development. In 80% of the cases the majority of problems can be traced back to the lack of a sufficient clear definition of project requirements.

In extreme non-programming the process of developing software systems is the following: First the customers describe their needs to the analyst. The analyst translates the customers' description into a model. Then the machine generates a human-readable view of that model which allows customers to clear up any misunderstandings before going further. A model can keep changing until the description is acceptable to the analyst and the customers. The next step is to automatically translate the model into code. In XNP, both the customers and the analyst share the same description of the problem; but they handle it in the way that is best suited for each.

Potential benefits of applying extreme non-programming are: (1) fast development for a software system; and (2) improved software quality.

7.2 New Challenges for Conceptual Modeling

Michael Carey in his ER2003 keynote address in Chicago issued a challenge to the conceptual modeling community to produce a simple conceptual model that (1) works well with XML and XML Schema; (2) abstracts well for conceptual entities and relationships; (3) scales to handle both large data sets and complex object interrelationships; (4) allows for queries and defined views via XQuery; and (5) accommodates heterogeneity. The conceptual model must work well with XML and XML Schema because XML is rapidly becoming the de facto standard for business data. Because conceptualizations must support both high-level understanding and high-level program construction, the conceptual model must abstract well. Because many of today's huge industrial conglomerations have large, enterprise-size data sets and increasingly complex constraints over their data, the conceptual model must scale up. Because XQuery, like XML, is rapidly becoming the industry standard, the conceptual model must smoothly incorporate both XQuery and XML. Finally, because we can no longer assume that all enterprise data is integrated, the conceptual model must accommodate heterogeneity. Accommodating heterogeneity also supports today's rapid acquisitions and mergers, which require fast-paced solutions to data integration.

References

[BBC91] M. Baldassari, G. Brouno, and A. Castella. PROTOB: An object-oriented case tool for modeling and prototyping distributed systems. *Software-Practice and Experience*, 21(8):822–844, 1991.

[BCM+03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, Massachusetts, 2003.

[Bec00] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 2000.

[BF90] B. Blanchard and W. Fabrycky. *Systems Engineering and Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1999.

[Bun77] M.A. Bunge. *Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World*. Reidel, Boston, Massachusetts, 1977.

[Bun79] M.A. Bunge. *Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems*. Reidel, Boston, Massachusetts, 1979.

[Che76] P.P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cho78] T.S. Chow. Testing design modeling by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[Dav92] A. M. Davis. Operational prototyping: a new development approach. *IEEE Software*, 9(5):70–78, 1992.

[DW04] D. Draheim and G. Weber. *Form-Oriented Analysis*. Springer Verlag, Berlin, Germany, 2004.

[EKW92] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-oriented Systems Analysis: A Model-Driven Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Gri82] R. Griffith. Three principles of representation for semantic networks. *ACM Transactions on Database Systems*, 7(3):417–422, September 1982.

[Gru93] T. Gruber. A translation approach to providing portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Hab91] N. Habra. Computer-aided prototyping: transformational approach. *Information and Software Technology*, 33(9):684–697, 1991.

[Hal01] T. Halpin. *Information Modeling and Relational Databases*. Morgan Kaufman, San Francisco, California, 2001.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, June 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[HPSH03] I. Horrocks, P.F. Patel-Schneider, and F.V. Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Web Semantics*, 1(1):7–26, 2003.

[JEW95] R.B. Jackson, D.W. Embley, and S.N. Woodfield. Developing formal object-oriented requirments specifications: A model, tool and technique. *Information Systems*, 20(4):273–289, 1995.

[Jon91] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, London, England, 1991.

[LG90] D.B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, Reading, Massachusetts, 1990.

[Min75] M. Minsky. A framework for representing knowledge. In P.H. Winston, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 211–217. McGraw-Hill, New York, New York, 1975.

[Pet77] J. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–52, 1977.

[Pre01] R.S. Pressman. *Software Engineering a Practitioner's Approach*. McGraw-Hill, Burr Ridge, Illinois, 2001.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersy, 1989.

[TSSO04] O. Turetken, D. Schuff, R. Sharda, and T.T. Ow. Supporting systems analysis and design through fisheye views. *Communication of the ACM*, 47(9):72–77, September 2004.

[TYF86] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[WW90] Y. Wand and R. Weber. An ontological model of an information system. *IEEE Transactions on Software Engineering*, 16(11):1282–1292, November 1990.