Representing and Using Non-Functional Requirements: A Process-Oriented Approach

John Mylopoulos, Lawrence Chung and Brian Nixon

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract The paper proposes a comprehensive framework for representing and using non-functional requirements during the development process. The framework consists of five basic components which provide for the representation of non-functional requirements in terms of interrelated goals. Such goals can be refined through refinement methods and can be evaluated in order to determine the degree to which a set of non-functional requirements is supported by a particular design. Evidence for the power of the framework is provided through the study of accuracy and performance requirements for information systems.

1 Introduction

The complexity of an information system is determined partly by its functionality — i.e., what the system does — and partly by global requirements on its development or operational costs, performance, reliability, maintainability, portability, robustness and the like. These *non-functional requirements*[1] play a crucial role during system development, serving as selection criteria for choosing among myriads of decisions. Errors of omission or commission in laying down and taking properly into account such requirements are generally acknowledged to be among the most expensive and difficult to correct once the information system has been completed. Surprisingly, non-functional requirements have received little attention by researchers and are definitely less well understood than other, less critical factors in software development. As far as software engineering practice is concerned, they are generally stated only informally during requirements analysis, are often contradictory, difficult to enforce during software development and to validate for the user once the final system has been built. The only glimmer of technical light in an otherwise bleak landscape originates in technical work on software quality metrics that allow the quantification of the degree to which a software system meets non-functional requirements [26, 5, 3].

There is not a formal definition or a complete list of non-functional requirements. In a report published by the Rome Air Development Center (RADC) [7], non-functional requirements ("software quality attributes" in their terminology) are classified into consumer-oriented (or *software quality factors*) and technically-oriented attributes (or *software quality criteria*). The former refers to non-functional requirements observable by the consumer, such as efficiency, correctness and interoperability. The latter addresses system-oriented requirements such as anomaly management, completeness and functional scope. Table 1.1 shows the RADC consumer-oriented attributes. The non-functional requirements listed in the table apply to all software systems. However, additional requirements may apply for special classes of software. For instance, precision would be an important non-functional requirement for a numerical analysis software package, while accuracy (of maintained information) might feature prominently during the development of an information system.

Two basic approaches characterize the formal treatment of non-functional requirements and we shall refer to them as *product-oriented* and *process-oriented*. The first attempts to develop formal definitions of non-functional requirements so that a software system can be evaluated as to the degree to which it meets its requirements. For example, measuring software visibility may include, among other things, measuring the amount of branching in a software system. This might be achieved globally with a criterion such as: "There shall be no more than X branches per 1,000 lines of code" or locally with a criterion such as "There shall be no more than Y% of system modules that violate the above criterion."

[1] Also referred to as *constraints* [41], *goals* [31] and *quality attributes* [26] in the literature.

Acquisition Concern	User Concern	Quality Attribute
Performance — How well does it function?	How well does it utilize a resource?	Efficiency
	How secure is it?	Integrity
	What confidence can be placed in what it does?	Reliability
	How well will it perform under adverse conditions?	Survivability
	How easy is it to use it?	Usability
Design — How valid is the design?	How well does it conform to the requirements?	Correctness
	How easy is it to repair?	Maintainability
	How easy is it to verify its performance?	Verifiability
Adaptation — How adaptable is it?	How easy is it to expand or upgrade its capability or performance?	Expandability
	How easy is it to change?	Flexibility
	How easy is it to interfere with another system?	Interoperability
	How easy is it to transport?	Portability
	How easy is it to convert for use in another application?	Reusability

Table 1.1 The RADC software quality consumer-oriented attributes. [26]

The product-oriented approach has received almost exclusive attention in the literature and is nicely overviewed in [26]. Earlier work by Boehm et al. [5] considered quality characteristics of software, noting that designer-awareness alone improved the quality of the final product. Also supporting a quantitative approach to software quality, Basili and Musa [3] advocate models and metrics of the software engineering process from a management perspective. It is interesting that Hauser et al. [21] provide a methodology for reflecting customer attributes in different phases of automobile design.

An alternative approach, explored in this paper, is to develop techniques for justifying design decisions during the software development process. Instead of evaluating the final product, the emphasis here is on trying to rationalize the development process in terms of non-functional requirements. Design decisions may affect positively or negatively particular non-functional requirements. These positive and negative dependencies can serve as basis for arguing that a software system indeed meets a certain non-functional requirement or explaining why it does not.

Orthogonally, treatments of non-functional requirements can be classified into *quantitative* and *qualitative* ones. Most of the product-oriented approaches alluded to earlier are quantitative in the sense that they study quantitative metrics for measuring the degree to which a software system satisfies a non-functional requirement. The process-oriented treatment proposed here, on the other hand, is definitely qualitative, adopting ideas from qualitative reasoning [1]. It should be acknowledged that a process-oriented treatment of non-functional requirements need not be qualitative. Indeed, one could imagine quantitative measures for, say, software visibility that can be used as the system is being developed to offer advance warning that non-functional requirements are not being met. Qualitative techniques were chosen here primarily because it was felt that the problem of quantitatively measuring an incomplete software system is even harder than that of measuring the final product.

Of course, neither product-oriented quantitative metrics nor process-oriented qualitative measures have a monopoly on properly treating non-functional requirements. They are best seen as complementary, both contributing to an evolving comprehensive framework for dealing with non-functional requirements.

Two sources of ideas were particularly influential on our work. The first involves recent work on decision support systems, such as that described in [28, 29] and [19]. Lee's work, for example, adopts an earlier model for representing design rationale [38] and extends it by making explicit the goals presupposed by arguments. The work reported here can be seen as an attempt to adopt this model to the representation and use of non-functional requirements. The second source of ideas is the DAIDA environment for information system development [23] which has provided us with a comprehensive software development framework covering both notations for requirements modelling, design, implementation and decision support, as well as a starting point on how the treatment of non-functional requirements might be integrated into that framework. Users of the DAIDA environment are offered three languages through which they can elaborate requirements, design and implementation specifications. In developing a design specification, the user consults and is constrained by corresponding requirements specifications. Likewise, the generation of an implementation is guided by a

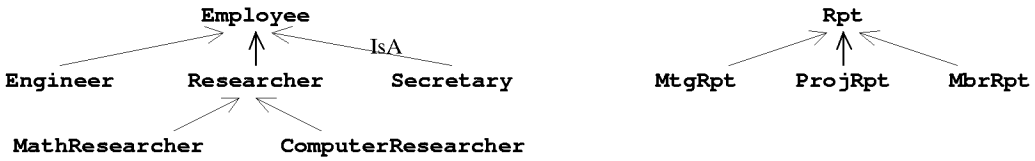


Figure 1.1: Employee and Report Hierarchy.

corresponding design specification. *Dependency links* represent design decisions and relate implementation objects to their design counterparts and design objects to their requirements counterparts. The framework proposed in this paper focuses on these dependency links and how they can be justified in terms of non-functional requirements. An early description of the framework and an account of how it relates to DAIDA can be found in [12].

The example used throughout this paper is an *expense management system* for a hypothetical research project, similar to the one used in [6]. According to the example, project members from organizations based in different countries, register for and attend various meetings. They then submit their expense summaries to an expense management system, which maintains all such information and generates expense reports for each member, meeting and project. As shown in Figure 1.1, there are several kinds of employees, including secretaries, engineers and researchers, who are in turn classified into computer researchers, math researchers, and so on.

Establishment of the framework is achieved in two steps. Firstly, the framework is presented in Section 2. The presentation includes motivation, the framework's structure and short suggestive examples. This framework is then elaborated and illustrated in the following two sections by examining its application respectively to accuracy and performance requirements for information systems. The final section summarizes the contributions of this research and presents a number of open questions and directions for further research.

2 Representing Non-Functional Requirements: A Process-Oriented Framework

Formally, the proposed framework consists of five major components[2]: a set of *goals* for representing non-functional requirements, design decisions and arguments in support of or against other goals; a set of *link types* for relating goals or goal relationships (hereafter links) to other goals; a set of generic *methods* for refining goals into other goals; a collection of *correlation rules* for inferring potential interactions among goals; finally, a *labelling procedure* which determines the degree to which any given non-functional requirement is being addressed by a set of design decisions. The examples throughout this section concentrate on accuracy and to a lesser extent operating cost requirements for information systems.

During the design process, goals are organized into a goal graph structure, very much in the spirit of AND/OR trees used in problem solving [34]. Unlike traditional problem solving and planning frameworks, however, goals representing non-functional requirements can rarely be said to be "accomplished" or "satisfied" in a clearcut sense. Instead, different design decisions contribute positively or negatively towards a particular goal. Accordingly, for the rest of the discussion we will speak of goal *satisficing* [42] [3] to suggest that generated software is expected to satisfy within acceptable limits, rather than absolutely, non-functional requirements.

[2] An earlier version of portions of this and the next section have appeared in [13].

[3] [42] actually uses the term to refer to decision methods that look for satisfactory solutions rather than optimal ones. The term is adopted here in a broadened sense since in the context of non-functional requirements, even the notions of a solution or optimality of a solution may be unclear.

2.1 Goals

The space of goals includes three mutually exclusive classes, namely, *non-functional requirements goals (NFR goals), satisficing goals* and *argumentation goals*. In general, each goal will have an associated *sort* and zero or more *parameters* whose nature depends on the goal sort. For example, an operating cost requirement might have as parameter a desired upper bound on the annual operating costs of the system under development. Sorts may be further subdivided into subsorts, representing special cases for each goal class. For instance, the *Performance* sort may have subsorts *TimePerformance* (or simply *Time*) and *SpacePerformance* (or simply *Space*), representing respective time and space performance requirements on a particular system. *Goals, NFRGoals, SatGoals* and *ArgGoals* will refer respectively to the set of all possible goals, NFR goals, satisficing goals and argumentation goals.

1. Non-functional requirements goals. The sorts for such goals range over the different categories of such requirements, including accuracy, security, development, operating or hardware costs and performance. For our expense management system, suppose that it is expected of the system under development to maintain accurately employee data. Such a goal might be represented by:

$$\mathit{Accuracy}[\texttt{attributes(Employee)}]$$

where *Accuracy* is the goal sort and the parameter of `attributes(Employee)` evaluates to the set of all attributes associated with the data class `Employee`. The interpretation of this goal is that instances of the attributes of the data class `Employee`, i.e., all attributes of employees, ought to be maintained accurately in the system's database. As another example, it may also be expected that the system under development make minimal demands on manpower. This can be treated as an operating cost requirement and since there are several contributing factors to operating costs (manpower, maintenance, etc.), this requirement might be represented as $\mathit{OperatingCost}[\texttt{manpower}]$.

2. Satisficing goals. These are also sorted and parameterized. In this case, however, the sorts range over different categories of design decisions that might be adopted in order to satisfice one or more non-functional requirements goals. The parameters associated with each sort, again, depend on the nature of the corresponding satisficing goal. For instance, one way to satisfice the accuracy goal mentioned earlier might be to validate all employee data entered into the system. This can be represented as a satisficing goal:

$$\mathit{Validation}[\texttt{attributes(Employee)}],$$

where *Validation* is the goal sort and `attributes(Employee)` is as before. This goal, in turn, might be refined into another satisficing goal,

$$\mathit{ValidatedBy}[\texttt{JohnWong},\texttt{attributes(Employee)}],$$

representing the situation that `JohnWong` will be doing the validation.

3. Argumentation Goals (or **Arguments**). These always have the sort *Claim*, with subsorts *FormalClaim* and *InformalClaim*, representing formally or informally stated evidence or counter-evidence for other goals or goal refinements. Consider:

$$\mathit{FormalClaim}[\exists \texttt{e} : \mathit{ValidatedBy}[\texttt{e},\texttt{attributes(Employee)}] \wedge \mathit{EmpStatus}(\texttt{e},\texttt{Sec I})]$$

This argumentation goal supports the refinement from the goal of validating employee data to the one assigning `JohnWong` to the task, by claiming that class I secretaries will perform the validation. In contrast,

$$\mathit{InformalClaim}[\text{“Rigorous examination is recommended for publications by employees.”}]$$

is an informally-stated argumentation goal supporting the previous argumentation goal by pointing out why class I secretaries should validate employee data.

2.2 Link Types

As indicated earlier, design proceeds by refining one or more times each goal, the parent, into a set of other goals, the offspring. Unlike AND/OR goal trees, where the relationship between a collection of offspring

and their parent can only be *AND* or *OR*, in our proposed framework there can be several different types of relationships or *link types* describing how the satisficing of the offspring (or failure thereof) relates to the satisficing of the parent goal. The need for at least some link types is evidenced in [5] which states that some quality characteristics are necessary, but not sufficient, for achieving others. Boehm et al. then use a four-grade scale to correlate each quality metric with quality attributes in the final product.

Links may relate a parent goal to one or several of its offspring. In fact, links may also be used to relate other links to argumentation goals, to indicate that an argument offers positive or negative support for a particular refinement of a goal. Thus, links too need to be satisficed either through a formal refinement process or through arguments provided by the designer.

Let *Links* denote the set of all links and *satisficed* be a predicate which is true of satisficed goals or links and false of others. Also, let *denied* be a predicate which is true of goals and links that have been shown unsatisficeable ("unsolvable" in problem solving terminology [34]). If

$$Propositions \quad = \quad Goals \cup Links,$$

then *satisficed* and *denied* are predicates taking a proposition as argument.

Sometimes a proposition will be found to be satisficeable — thanks to one refinement — and deniable — thanks to another. For instance, the accuracy goal for employee data might be satisficeable thanks to a validation procedure adopted for all such data, but deniable because of a user interface that permits general access to this information. To deal with such conflicting cases, we need to distinguish between a proposition being satisficed or denied, on one hand, and a proposition being *potentially* satisficeable or deniable thanks to some refinement on the other. Accordingly, two more predicates, *satisficeable* and *deniable* are introduced to deal with the latter case.

The set of logical types to be used for links is presented below. For each type, axioms are provided which formalize its semantics in terms of the predicates just introduced:

$AND:$ $\quad Propositions \times 2^{Propositions}.$

$$satisficed(G_1) \wedge satisficed(G_2) \wedge \ldots \wedge satisficed(G_n) \ \wedge \ satisficed(AND(G_0, \{G_1, G_2, \ldots, G_n\})) \longrightarrow satisficeable(G_0)$$

$$satisficed(AND(G_0, \{G_1, G_2, \ldots, G_n\}) \wedge (denied(G_1) \vee denied(G_2) \vee \ldots \vee denied(G_n)) \longrightarrow deniable(G_0)$$

$OR:$ $\quad Propositions \times 2^{Propositions}.$

$$denied(G_1) \wedge denied(G_2) \wedge \ldots \wedge denied(G_n) \ \wedge \ satisficed(OR(G_0, \{G_1, G_2, \ldots, G_n\})) \longrightarrow deniable(G_0)$$

$$satisficed(OR(G_0, \{G_1, G_2, \ldots, G_n\}) \wedge (satisficed(G_1) \vee satisficed(G_2) \vee \ldots \vee satisficed(G_n)) \longrightarrow satisficeable(G_0)$$

$sup:$ $\quad Propositions \times Propositions.$

$$satisficed(G_1) \ \wedge \ satisficed(sup(G_0, G_1)) \longrightarrow satisficeable(G_0)$$

$sub:$ $\quad Propositions \times Propositions.$

$$denied(G_1) \ \wedge \ satisficed(sub(G_0, G_1)) \longrightarrow deniable(G_0)$$

The link type *sub* is also intended to convey the sense that G_1 contributes partially to the satisficing of G_0. This can be expressed as follows: If $satisficed(sub(G_0, G_1))$ then there exist propositions $G_2, \ldots, G_n$ such that

$$\neg(satisficed(G_2) \wedge \ldots \wedge satisficed(G_n) \longrightarrow satisficeable(G_0))$$

but

$$satisficed(G_1) \wedge satisficed(G_2) \wedge \ldots \wedge satisficed(G_n) \ \wedge \ satisficed(sub(G_0, G_1)) \longrightarrow satisficeable(G_0)$$

In words, if G_1 is a *sub*(proposition) of G_0 then there exist propositions $G_2, \ldots, G_n$ which cannot achieve the satisficing of G_0 without the contribution of G_1.

Two additional link types are introduced to represent negative influences of one goal on another.

$-sup$: $Propositions \times Propositions.$

$satisficed(G_1) \ \wedge \ satisficed(-sup(G_0, G_1)) \longrightarrow deniable(G_0)$

$-sub$: $Propositions \times Propositions.$

$denied(G_1) \ \wedge \ satisficed(-sub(G_0, G_1)) \longrightarrow satisficeable(G_0)$

$If \ -sub(G_0, G_1)$ *then there exist* $G_2, \ldots, G_n$ *such that*

$\neg(satisficed(G_2) \wedge \ldots \wedge satisficed(G_n) \ \wedge \ satisficed(-sub(G_0, G_1)) \longrightarrow deniable(G_0))$

but

$satisficed(G_1) \wedge satisficed(G_2) \wedge \ldots \wedge satisficed(G_n) \ \wedge \ satisficed(-sub(G_0, G_1)) \longrightarrow deniable(G_0)$

In words, if G_1 is a *negative sub*(proposition) of G_0 then denial of G_1 leads to the satisficing of G_0 and satisficing of G_1 contributes to the denial of G_0.

Finally, it is useful to define the *eql (equivalent)* link type in terms of the link types introduced here:

eql : $Propositions \times Propositions.$

$eql(G_0, G_1) \equiv sup(G_0, G_1) \ \wedge \ sup(G_1, G_0) \ \wedge \ sub(G_0, G_1) \wedge \ sub(G_1, G_0)$

At times, it may be hard to determine *a priori* the logical relationship between a set of offspring and their parent goal without further expansion of the goal graph. For example, the designer may see that a certain hiring policy for technical staff is relevant, without being sure of its impact on a particular goal, say, in justifying the assignment of a class I secretary to the task of validating employee data. This situation is accommodated through three variations of an undetermined link type:

und : $Propositions \times Propositions.$

$und(G_0, G_1)$ *indicates the possible presence of positive or negative influence between* G_0 *and* G_1.

Likewise, $+und$ and $-und$ indicate respectively possible positive or negative influence between two propositions.

2.3 Methods

Goals may be refined by the designer, who is then responsible for satisficing not only the goal's offspring but also the refinement itself represented as a link. Alternatively, the framework provides *goal refinement methods* (methods for short) which represent generic procedures for refining a goal into one or more offspring, such as:

> "To maintain accurately data about class `x`, you need to maintain accurately data about all relevant subclasses of `x`."

Every such refinement is represented in terms of a link having one of the types of the previous section and which is considered satisficed.

Generally, a method has the form

$\mathtt{x_1}/\mathtt{C_1}, \mathtt{x_2}/\mathtt{C_2}, \ldots \mathtt{x_n}/\mathtt{C_n}$: $SelP(\mathtt{x_1}, \mathtt{x_2}, \ldots \mathtt{x_n}) \mid$

$G_0(\mathtt{x_1}, \ldots \mathtt{x_n}) \stackrel{L}{\longrightarrow} \{G'(\mathtt{x_1}, \ldots \mathtt{x_n}) \mid For\ all\ G'\ such\ that\ Pred(G', \mathtt{x_1}, \ldots \mathtt{x_n}))\}$

Here G_0 represents the parent goal, predicate $Pred$ determines the set of offspring while L is the link type relating G_0 to its offspring. The refinement of G_0 through a method is subject to the method's selection criterion, $SelP$, consisting of a Boolean expression with free variables $\mathtt{x_1}, \mathtt{x_2}, \ldots, \mathtt{x_n}$. These are bound to objects of type $\mathtt{C_1}, \ldots, \mathtt{C_n}$ respectively when the method is applied.

There are three types of goal refinement methods, corresponding to the three types of goals introduced earlier:

1. Goal Decomposition Methods. These are usually *AND* decomposition methods of the form $SelP : G \xrightarrow{AND} \{G_1, G_2, \ldots, G_n\}$ used to decompose a goal G into an *AND* set of offspring $G_1, G_2, \ldots, G_n$. For instance, the following method decomposes a goal having a class as argument into goals having as arguments its immediate specializations:

$$\texttt{x/Class}: \quad G[\texttt{x}] \xrightarrow{AND} \{G[\texttt{x}_\texttt{i}], \mid (\texttt{x}_\texttt{i}\ isA\ \texttt{x}) \ \wedge \ \forall \texttt{x}_\texttt{j}\ [((\texttt{x}_\texttt{i}\ isA\ \texttt{x}_\texttt{j}) \wedge\ (\texttt{x}_\texttt{j}\ isA\ X)) \Rightarrow ((\texttt{x}_\texttt{j} = \texttt{x}_\texttt{i}) \vee (\texttt{x}_\texttt{j} = \texttt{x}))]\}$$

Since there are three specializations of `Employee` in our example, the accuracy goal *Accuracy*[`attributes(Employee)`] (abbreviated as A[`attributes(Employee)`]) can be refined using the subclass goal decomposition method:

$$A[\texttt{attributes(Employee)}] \xrightarrow{AND} \{A[\texttt{attributes(Researcher)}], \ \ldots, \ A[\texttt{attributes(Secretary)}]\}$$

In Figure 2.1, offspring are shown underneath the parent goal. Link types are sometimes omitted from figures. Now each of these goals needs to be satisficed in turn. Likewise, satisficing the goal of A[`attributes(Researcher)`] requires that all attributes of Researcher be maintained accurately. This decomposition can be accomplished by a method of the form:

$$\texttt{x/Class} : A[\texttt{attributes(x)}] \xrightarrow{AND} \{A[attr(\texttt{x})] \mid attr \in \texttt{attributes(x)}\}$$

This method leads to the following further decomposition of A[`attributes(Researcher)`]. Assuming that research employees have attributes `degree` and `publ` (publications), in addition to those of Employee,

$$A[\texttt{attributes(Researcher)}] \xrightarrow{AND} \{A[\texttt{Researcher.name}], \ldots, A[\texttt{Researcher.degree}], \ A[\texttt{Researcher.publ}]\}$$

2. Goal Satisficing Methods. Such methods refine a goal into a set of satisficing goals, thereby committing the design that is being generated to particular design decisions. Returning to our example, there may be two satisficing methods offered for the goal A[`Researcher.publ`]. If the publication record of each researcher is obtained from existing databases, the accuracy of this information might be ensured through periodic auditing of those databases. If, on the other hand, these data are fed directly by the employee in question, a method may call for the validation of the data by the employee's manager:

$$\texttt{i/InformationItem}: A[\texttt{i}] \xrightarrow{+und} Audit[\texttt{i}]$$

$$\texttt{i/InformationItem}: A[\texttt{i}] \xrightarrow{sup} Validation[\texttt{i}]$$

Using these methods, A[`Researcher.publ`] can be refined to *Audit*[`Researcher.publ`] or *Validation*[`Researcher.publ`] through $+und$ and *sup* links respectively. Note that the designer may later change the type of the $+und$ link once the design has proceeded further and it can be determined that auditing indeed leads to more accurate publication data. Clearly, selection of one of the two alternatives leads to very different types of user interfaces for the system under development. In particular, if validation is selected, all publication information will have to be confirmed by another person, while auditing calls for the inclusion of an audit requirement on the database from which publication data are imported.

3. Argumentation methods. These methods refine a goal or a link into an argumentation goal, thereby indicating evidence/counter-evidence, in terms of arguments, for the satisficing of a goal. For instance, a formal claim consisting of a conjunction could be refined into claims of each conjunct related to the parent through an AND link.

Figure 2.1 illustrates the goal structure that might be generated by the simple example we have been introducing piecemeal. In the bigger picture of information system development, a source object, say a component of a requirements specification, is mapped into one (or possibly several) target object(s), say components of a design specification [13]. The dependencies among these objects are shown through dependency links on the left- and right-hand sides of Figure 2.1. The use of the goal structure generated by the designer from non-functional requirements, possibly with the help of methods, is intended to help her *select* among alternatives and *justify* her design decisions. She can selectively focus attention, thus controlling goal structure expansion.

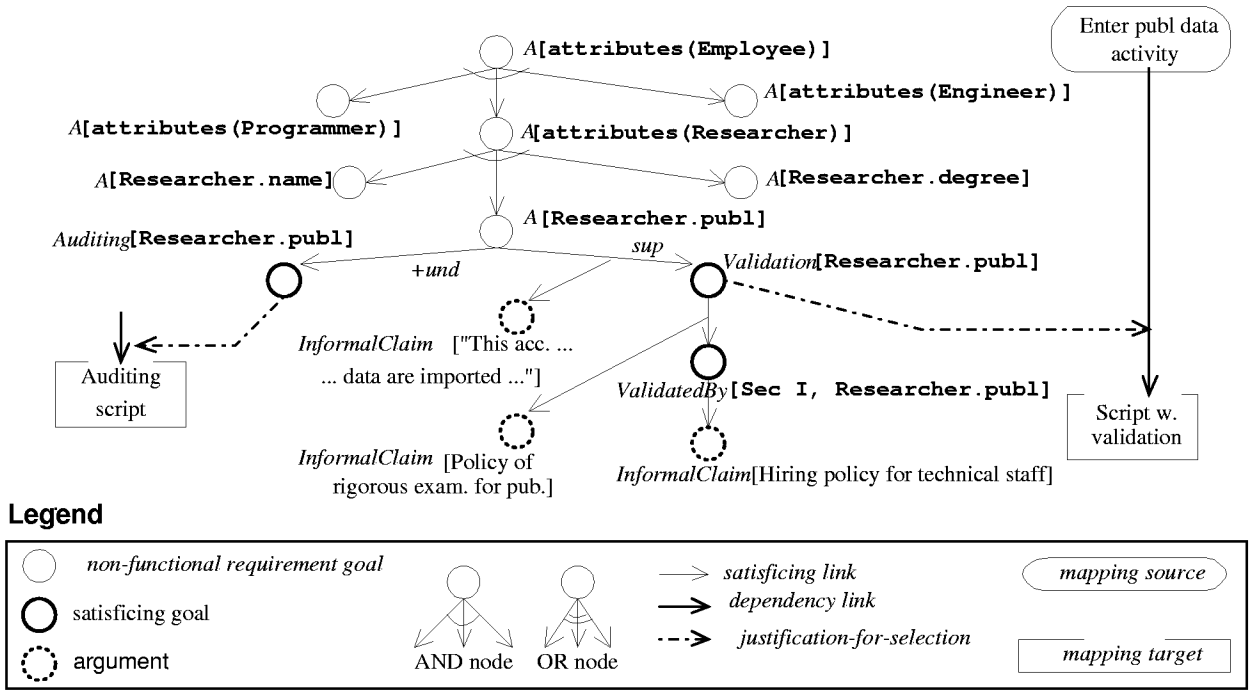


Figure 2.1: Goal graph structure for accurate employee attributes.

2.4 Correlation Rules

As indicated earlier, the non-functional requirements set down for a particular system may be contradictory. For instance, having built-in procedures for validating or auditing the data managed by the information system in general requires additional manpower thereby interfering with the operating cost requirement, *OperatingCost*[`manpower`]. Guidance is needed by the designer in discovering such implicit relationships and in selecting the satisficing goals that best meet a set of given NFR goals. This is achieved either through external input by the designer herself or through the representation of generic interactions between goals through *correlation rules*.

Consider a satisficing goal whereby the system under design will offer an interface for "casual" users, say, all company employees, who wish to query or update the system's database.

$$OperatingCost[\texttt{manpower}] \xrightarrow{sub} CasualUserInterface[\texttt{Employee,Database}]$$

Unfortunately, making the database readily available to all employees is likely to lead to data inaccuracy, thereby interfering with accuracy goals. This can be expressed, for example, by a rule such as:

$$CasualUserInterface[\texttt{e},\texttt{i}] \wedge cardinality(\texttt{e}) > 5 \wedge A[\texttt{i}'] \wedge \texttt{i}' \subset \texttt{i} \longrightarrow -sub(A[\texttt{i}'], CasualUserInterface[\texttt{e},\texttt{i}])$$

This rule can be used to infer a *-sub* link between the goals *CasualUserInterface*[`Employee, Database`] and *A*[`attributes(Employee)`], assuming that `attributes(Employee)` $\subset$ `Database` and that there are more than five employees.

Likewise, consider a security goal (with sort S) discouraging secretaries from accessing research publication data. Now a validation goal which is positive for *A*[`Researcher.publ`], with a class I secretary as validator, would also contribute negatively to such a security goal, and vice versa:[4]

$$(S[\texttt{i},\texttt{e},\texttt{accessCond}] \wedge ValidatedBy[\texttt{e}',\texttt{i}'] \wedge (\texttt{i} \subseteq \texttt{i}') \wedge isA(\texttt{e},\texttt{e}') \wedge HigherClassification(\texttt{e},\texttt{e}') \wedge \texttt{accessCond}) \longrightarrow (-sup(S[\texttt{i},\texttt{e},\texttt{accessCond}], Validation[\texttt{i}']) \wedge -sup(Validation[\texttt{i}'], S[\texttt{i},\texttt{e},\texttt{accessCond}]))$$

[4] The representation of security requirements is adopted from [20].

Finally, consider the case where two satisficing goals interfere with each other because of dependence on a critical resource. This competition may be *synergistic* or *antagonistic*, leading respectively to positive or negative argumentation. For instance, two unrelated goals calling for auditing and validation of information may influence each other positively (through **sub** links if there is no one on staff assigned to either task, because they jointly suggest the hiring of of personnel data may not individually justify hiring additional staff. If, however, the argumentative structure indicates that they can share an agent, one new staff member may be hired for the two tasks.

$$Validation[\mathtt{i}] \wedge Audit[\mathtt{i}'] \quad \wedge \quad \neg\exists\, \mathtt{e}, \mathtt{e}'/\mathtt{Employee} : ValidatedBy[\mathtt{e}, \mathtt{i}] \wedge AuditedBy[\mathtt{e}', \mathtt{i}'] \longrightarrow sub(Validation[\mathtt{i}], Audit[\mathtt{i}']) \wedge sub(Audit[\mathtt{i}'], Validation[\mathtt{i}])$$

It is now possible to describe the expansion procedure which starts with a set of NFR goals and iteratively expands them into a goal graph structure. Throughout the expansion, the system maintains a list of all propositions that are to be refined, called *Open*, while the list *Closed* includes all propositions that have been completely refined.

Once a proposition has been selected from *Open* for refinement, the designer chooses whether she wants to propose a refinement or apply one of the available methods. Carrying out a chosen refinement involves creating propositions for the offspring and the newly-created link and adding each to *Open*. Correlation links are then introduced for the new propositions, using both the designer's judgement and correlation rules in the system. This process is repeated for the chosen proposition until there are no more refinements the system or the designer can offer. At this time, the proposition is placed on the *Closed* list and another open proposition is selected.

A second alternative for proposition refinement is to simply label the proposition *satisficed* or *denied*. Such labelling may come about either because of input from the designer or because of the use of a method during proposition refinement.

2.5 The Labelling Procedure

Given a partially constructed goal graph structure, the labelling procedure determines the status of each node on the graph through the assignment of a label. A node or link of the graph is labelled *satisficed* if it is satisficeable and not deniable; *denied* if it is deniable but not satisficeable; *conflicting* if it is both satisficeable and deniable; and *undetermined* if it is neither. These labels are denoted respectively by *S, D, C* and *U*. They are similar to those in [16] and generally ones used in qualitative reasoning frameworks [1]. The *U* label, in particular, is intended to represent situations where either there is both positive and negative support, albeit inconclusive, for a given goal, or there is neither positive nor negative support.

The labelling algorithm consists of two basic steps. For each proposition **P** on a given goal graph, the algorithm first computes the individual effect of each satisficed outgoing link. Secondly, the individual effects of all outgoing links are combined into a single label taking one of the four possible values mentioned earlier.

Given the open-ended nature of the argumentation process (i.e, the premise built into this framework that only some of the relevant knowledge is formally represented, the rest remaining with designers) the framework calls for an interactive labelling procedure where the designer may be asked to step in and determine the appropriate label for a particular proposition having supporting but inconclusive evidence. For this reason, the labels characterizing the influence of one set of offspring towards a parent include *S, D, C,* and *U*, as mentioned before, but also U^- and U^+ representing respectively inconclusive positive or negative support for a parent. Moreover, ? indicates a situation where the designer is to determine the label that characterizes the contribution of a proposition towards another. Note that the labels U^-, U^+ and ? are introduced by the first step of the labelling algorithm and are eliminated by the second when the set of all contributions from all outgoing links associated with a given proposition are combined into a single label, *S, D, C* or *U*.

Table 2.1 shows the propagation rules along different link types (always from offspring to parent). According to these rules, *sup* propagates *S* while *sub* propagates *D*; *-sup* inverts an *S* label into a *D* and *-sub* inverts a *D* label into a *S* one.

$label_{source}$	link type				
	sub	*sup*	*-sub*	*-sup*	*und*
S	U^+	S	U^-	D	U
D	D	U^-	S	U^+	U
C	?	?	?	?	U
U	U	U	U	U	U

Table 2.1 The *individual effect* of source label upon its destination label.

The propagation rules for *AND* and *OR* links are based on the ordering of labels $S \geq U, C \geq D$ and is defined as follows:

$$Assuming \quad AND(G_0, \{G_1, G_2, \ldots, G_n\}) \quad then\ label(G_0) = \min_i(label(G_i))$$

$$Assuming \quad OR(G_0, \{G_1, G_2, \ldots, G_n\}) \quad then\ label(G_0) = \max_i(label(G_i))$$

Once all contributed labels have been collected for a given proposition, the second step of the labelling procedure combines them into a single label. Assuming that $\mathtt{L}$ is the bag [5] contributed to a given proposition, consisting of labels from the set $\{S, D, C, U, U^-, U^+\}$, the U^+ and U^- labels are first combined by the designer into one or more S, D, C and U labels. The resulting set of labels is then combined into a single one, by choosing the minimal element, $\min_{l \in \mathtt{L}}(l)$, and assuming a label ordering $S, D \geq U \geq C$.

It is interesting to compare our labelling procedure with those of truth maintenance systems (TMSs) [15, 17]. They record and maintain beliefs, their justifications and assumptions, while distinguishing facts from defeasible beliefs, which are either accepted or rejected. As with TMSs, our graph labelling procedure recursively propagates values of offspring to parents. However, our procedure is not automatic, but interactively allows the designer to deal with inconclusive evidence. While we have *AND* and *OR*, comparable to TMS conjunction and disjunction, our link types have additional values, all of which are inputs to computing *individual effect* in our first step. In applying the propagation rules of Table 2.1, *links*, which are not included in TMS beliefs, must be *satisficed*. Unlike TMSs, we then *combine* individual effects of label values including qualitative (*conflicting*) and open-ended (*undetermined*) ones, using a label ordering in the second step.

3 Dealing with Accuracy Requirements

A major consideration in building an information system is the degree to which its design encourages accuracy of the information being managed. For example, a system which allows users to update information in their own files may be user-friendly, but one will not have confidence in the information it contains. There are many ways to promote accuracy requirements for an information system. Restricting access to resources is only one such technique.

Within our framework, treating accuracy requirements as goals offers directional guidance for the overall design process. In particular, accuracy requirements are used below as criteria for selecting a particular design in order to address elements of a given functional requirement.

3.1 Goals of Accuracy Requirements

The goals of accuracy requirements have *Accuracy* as the sort and `InformationItem` (abbreviated `Info`) as the parameter. They are expressed as *Accuracy*[`i`] (abbreviated as A[`i`]), where `i` is a collection of information items. Information items may be categorized into three types of propositions: i) that an entity in the system has the *property* of some class during some time interval; ii) that an entity in the system has an *attribute* with a certain value during some time interval; iii) that an object in the system, say a record, has one and only one corresponding *entity* in the application domain, say an employee. Accuracy requirements can then be expressed on collections of such information items, such as the employee *attributes* of Section 2 (See [12, 13] for this). In general, satisficing accuracy goals is understood in terms of the degree of confidence in the accuracy of information items maintained by the projected system.

[5] $\mathtt{L}$ is a bag because duplicate labels are useful; for instance several positive supporting links indicated by several U^+'s may be combined into an S label by the designer.

3.2 Goal Refinement Methods

3.2.1 Goal Decomposition Methods

We present below some examples of accuracy decomposition methods, to be used in the illustration of Section 3.4.

- *subclass* method: In order to establish the accuracy of a class `C` of information items, establish the accuracy of each immediate specialization, $\texttt{C}_\texttt{i}$, of `C`. This is a special case of the goal decomposition method mentioned in Section 2.3.

- *subset* method: To establish the accuracy of a set of information items, establish the accuracy of each subset of information items. Similarly, a *superset* method can be provided.

- *individualAttributes* method: To establish the accuracy of the attributes of a class of information items, establish the accuracy of each attribute of the class.

- *derivedInfo* method: To establish the accuracy of a set of information items, establish that the function which derives them is correctly designed and that all of the function's source parameters, currently in the system, are accurate.

- *attributeSelection* method: To establish the accuracy of an information item obtained by a sequence of attribute selections (e.g., `Joe.project.budget`), establish the accuracy of each information item obtained in the sequence (e.g., `Joe.project`, `Project.budget`).

- *conservation* method: To establish the accuracy of a collection of information items which can no longer be decomposed into information items currently in the system, establish i) their accuracy, when received by the system from some external agent, and ii) their correct internal manipulation by the system.

- *correctExternalManipulation* method: To establish the accuracy of information items upon receipt, establish *CorrectInfoFlow*, i.e., they were accurate when they were first transmitted by the original sender, and have subsequently been correctly manipulated until receipt by the system. *CorrectInfoFlow* is a sub-sort of *Correctness* goals which, unlike accuracy goals, are related to actions that induce certain results.

3.2.2 Goal Satisficing Methods

Taking the premise that the accuracy of information items depends entirely on the process in which they are manipulated within the system and its environment, accuracy satisficing goals alter *that* process[6]. Accuracy satisficing goals include *preventive*, *curative* and *precautionary techniques*. They affect the level of our confidence in the accuracy of information items.

Preventive accuracy satisficing goals detect and disallow inaccuracies, when information items are received by the system. Most of them require direct interaction between the system and agents in the application domain. They can be specialized by varying the agent who performs the needed task, the volume of information items, evidences attached, the time of processing and output, etc.:

- *confirmation*: The informant, either a machine or a person, double-checks the previously-submitted information item. This technique can be specialized: to *confirmation-via-identical-channel* if the confirmation and first transmission use the same channel; otherwise to *confirmation-via-distinct-channel* (e.g., via a daisy-channel).

- *verification*: A *verifier*, who is a co-worker of the sender of information item makes a duplicate entry of the item onto some medium in the system (e.g., via duplicate IBM key-entry operation). As with confirmation, verification can be specialized to *verification-via-identical-channel* or *verification-via-distinct-channel*.

[6] Martin[30], for instance, offers a glossary of techniques for improving accuracy.

- *validation*: A *validator* performs checking in the application domain, using certain records or procedural guidelines to ensure that the information item meets predetermined standards. The type and thoroughness of the checking can be reflected in specialized methods: *creation-validation* for directly contacting the information source, *experimentation* for re-testing the information item, etc.

- *audit*: An *accuracy auditor* uses procedures to periodically go through suspicious sampled information items.

- *consistency-checking*: To prevent frequently-occurring errors, the system enforces certain integrity constraints (e.g., check-sums incorporated into ISBNs).

Curative satisficing goals trace inaccuracies to their source, and provide for recovery from inaccuracies. *Precautionary* satisficing goals make information flow more reliable in terms of what is involved, such as senders, receivers, and communication channels.

3.2.3 Goal Argumentation Methods

These methods support or deny the use of accuracy satisficing goals and various refinements in terms of arguments. Examples include:

- *resource-assignment:* In performing a task for a satisficing goal, assign resources in the application domain. For example, one can support a refinement from a goal of validating expense summaries to one assigning a staff member to the task, by claiming that class I secretaries will perform the validation.

$$Validation[\texttt{Summary}] \xrightarrow{sup} FormalClaim[\exists \texttt{e} : ValidatedBy[\texttt{e}, \texttt{Summary}] \wedge EmpStatus(\texttt{e}, \texttt{Sec I})]$$

- *policy-manual-consultation:* When a question arises about the applicability of various types of methods, consult policy manuals in the application domain.

- *priority-based-selection:* Select a method among alternatives according to their relative priority. E.g., for a satisficing goal which is good for high-priority accuracy goal A but bad for goal B, the priority would be a positive argument for A but negative for B.

3.3 Correlation Rules

Accuracy satisficing goals, such as verification, usually contribute positively to accuracy goals (such as $A[\texttt{attribute(Researcher)}]$) provided the (verification) process is rapid. Otherwise, information items will become less timely. This perturbation is an example of a satisficing goal becoming negative. For example:[7]

$$VerifiedBy[\texttt{e}, \texttt{i}, \texttt{t}] \ \wedge \ Excessive(t) \wedge \ A[\texttt{i}'] \ \wedge \ \texttt{i}' \subseteq \texttt{i} \longrightarrow -sub(A[\texttt{i}'], Verification[\texttt{i}])$$

Verification may be negative for a security goal if the verifier is not allowed to access the information item to be verified.

A security satisficing goal (such as *Mutual-ID*) or a user-friendliness satisficing goal (such as *CasualUserInterface*) can be positive or negative for an accuracy goal. Consider *Mutual-ID*[`a:Agent`, `i:Info`, `p:Procedure`, `t:Time`]. To mutually ensure the identity of the agent `a`, attempting to access certain information items `i`, and the identity of the system process, both the agent and the system, during time interval `t`, go through a test procedure, `p`, which requires alternating queries and answers by the two (This is similar to the *challenge response* process [37]). This would be positive for accuracy goals if a malicious user, in the absence of mutual identification, would penetrate the system and falsify the information item.

Table 3.1 summarizes some of the correlations. Entries of the form: $< condition, orientation >$ mean "if the *condition* holds, then the relationship between the requirements goal and satisficing goal is given by the *orientation*."

[7] The time parameter (`t`) is omitted when not needed.

NFRGoal / **SatGoal**	*Accuracy*	*Security*
Verification	$< R_1,\ sub >, < R_2,\ -sub >$	$< R_4,\ -sub >$
Mutual-ID	$< R_3,\ sub >$	$< TRUE,\ sub >$
CasualUserInterface	$< TRUE,\ -sub >$	

R_0: $A[\mathtt{i}'] \ \wedge \ \mathtt{i}' \subseteq \mathtt{i}$
R_1: $VerifiedBy[\mathtt{e}, \mathtt{i}, \mathtt{t}] \ \wedge \ R_0$
R_2: $R_1 \ \wedge \ Excessive[\mathtt{t}]$
R_3: $Mutual-ID[\mathtt{e}, \mathtt{i}, \mathtt{p}, \mathtt{t}] \ \wedge R_0 \ \wedge \ Informant-ID-established[\mathtt{e}]$
R_4: $S[\mathtt{i}, \mathtt{e}, \mathtt{AccessCond}] \ \wedge \ VerifiedBy[\mathtt{e}', \mathtt{i}', \mathtt{t}] \ \wedge \ \mathtt{i} \subseteq \mathtt{i}' \ \wedge \ isA(\mathtt{e}, \mathtt{e}') \wedge \ HigherClassification(\mathtt{e}, \mathtt{e}') \ \wedge \ \mathtt{AccessCond}$

Table 3.1 Correlation of NFR Goal *(NFRGoal)* with Satisficing Goals (**SatGoal**).

The table is similar in spirit to the "relationship matrix" [21], which indicates, informally and without correlation rules, how much each engineering characteristic affects each customer quality requirement in terms of four types of values: strong positive, medium positive, medium negative or strong negative.

An accuracy satisficing goal can be synergistic or antagonistic with respect to another satisficing goal, for one or more types of non-functional requirements goals. Suppose a single channel can sometimes be shared for confirmation and verification. Now *confirmation-via-distinct-channel* and *verification-via-distinct-channel* are mutually synergistic if a new channel can be installed for shared use by the two, but mutually antagonistic if the channel is unshareable.

3.4 Illustration

Consider the example of research expense management system in Section 1. Now assume that $A[\mathtt{attributes(Rpt)}]$ is the root node of the goal tree representing an accuracy requirement, "all the attributes of expense reports should be accurate". The root goal can be refined with the *subclass* method into three offspring corresponding to the subclasses of `Rpt`, specified as part of functional requirements (See Figure 3.1):

$$A[\mathtt{attributes(Rpt)}] \xrightarrow{AND} \{A[\mathtt{attributes(ProjRpt)}],\ A[\mathtt{attributes(MtgRpt)}],\ A[\mathtt{attributes(MbrRpt)}]\}$$

Now each of these offspring needs to be satisficed. Focusing on the subgoal of A[`attributes(ProjRpt)`], the goal of $A[\mathtt{attributes(ProjRpt)}]$ is decomposed by the *individualAttributes* method in terms of the accuracy of the attributes.

$$A[\mathtt{attributes(ProjRpt)}] \xrightarrow{AND} \{A[\mathtt{ProjRpt.mon}], \ldots, A[\mathtt{ProjRpt.budgetLeft}]\}$$

The legend for the symbols is given in Figure 2.1 (When omitted, assume that the link type for satisficing and argumentation methods is *sup* in the remainder of this paper).

Focusing on $A[(\mathtt{ProjRpt.exp})]$, the designer indicates that `ProjRpt.exp` is a *derived* information item, where the derivation function, `f`, is shown in Figure 3.1. Thus, the *derivedInfo* decomposition method is instantiated: the function needs to be correctly designed and the parameters of the function should be accurate. Next the *subset* method is instantiated for the decomposition of $AccurateParameters[\mathtt{f}]$:

$$A[\mathtt{ProjRpt.exp}] \xrightarrow{AND} \{CorrectDerivFn[\mathtt{f}, \mathtt{ProjRpt.exp}],\ \ AccurateParameters[\mathtt{f}]\}$$
$$AccurateParameters[\mathtt{f}] \xrightarrow{AND} \{A[\mathtt{Exp.date}], \ldots, A[\mathtt{Exp.proj}]\}$$

Two competing alternatives (i.e., disjunctive refinements) are foreseen by the designer for the date the expense was incurred: it may come from either the expense reimbursement requests (by requiring the members to send their reimbursement request forms to the central management office), or the expense summary (by requiring the secretary to submit it directly):

$$A[\mathtt{Exp.date}] \xrightarrow{OR} \{A[\mathtt{Exp.reim-req.date}],\ A[\mathtt{Exp.summary.when}]\}$$

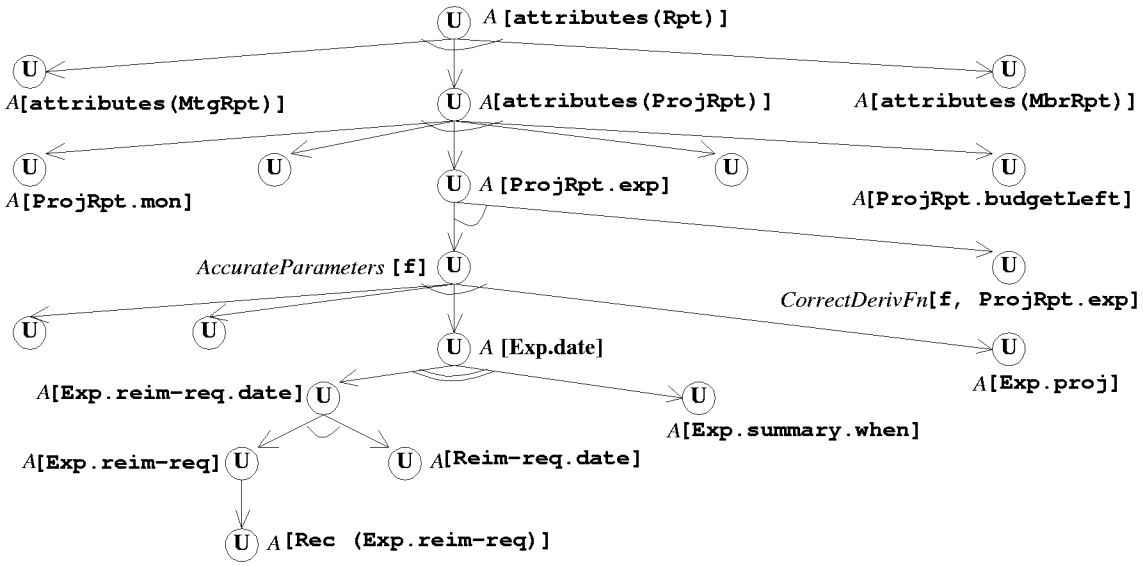
To explore the first alternative, the designer applies the *attributeSelection* method :

$$A[\mathtt{Exp.reim-req.date}] \xrightarrow{AND} \{A[\mathtt{Exp.reim-req}],\ A[\mathtt{Reim-req.date}]\}$$

To illustrate manipulation of information items, we introduce some method applications which were not shown in Figure 2.1. The designer indicates that `Exp.reim-req` should be received from an external agent. According to the *conservation* method, `Exp.reim-req` should be both accurate when received and correct when processed by the system:

$$A[\texttt{Exp.reim-req}] \xrightarrow{AND} \{A[\texttt{Rec(Exp.reim-req)}],\ CorrectProcessing[\texttt{Reim-req.date}]\}$$

When invoked by the designer, the labelling procedure assigns U (undetermined) to all goals, since there are no closed leaves. Although omitted, all the links in Figure 3.1 have S as their labels, since they are the results of generic-method applications.



where `f = ComputeAmount (Exp, Exp.date, Exp.proj, ProjRpt.mon, ProjRpt.proj)`

Figure 3.1: Goal graph structure with decompositions for accurate expense-reports attributes.

The designer uses the *correctExternalManipulation* method to refine the accuracy of the received information item (Figure 3.2):

$$A[\texttt{Rec(Exp.reim-req)}] \xrightarrow{AND} \{CorrectCreation[\texttt{Rec(Exp.reim-req)}], CorrectInfoFlow[\texttt{Rec(Exp.reim-req)}]\}$$

Unfortunately, ensuring the correct creation and subsequent transmissions of the item from the creator to the system is in many cases costly and impractical. Accordingly, the designer may resign himself to using some satisficing methods for A[`Rec (Exp.reim-req)`]. In selecting a method, the designer uses the argumentation method of *policy-manual-consultation*, *Designer's Consultation Guidelines (DCG)*. The designer regards the *validation* method to be appropriate:

$$CorrectInfoFlow[\texttt{Rec(Exp.reim-req)}] \xrightarrow{sup} Validation[\texttt{Exp.reim-req}]$$

Note how the method above (call it $validation_e$) is supported by a designer-supplied argument:

$$validation_e \xrightarrow{sup} InformalClaim[\text{"DCG : Careful examination is preferred for those materials that are directly related to issuing a cheque."}]$$

To satisfice the goal of validation, the designer again consults the *DCG* and discovers that class I secretary is one, but not the only, good class of candidate for carrying out the validation. Thus, a class I secretary is assigned (call the assignment, $assign_v$) and the assignment is supported by:

$$Validation[\texttt{Exp.reim-req}] \xrightarrow{eql} FormalClaim[ValidatedBy[\texttt{Sec I}, \ldots] \wedge \ldots]$$
$$assign_v \xrightarrow{sup} InformalClaim[\text{"DCG : For } \ldots, \text{ consider class I secretary."}]$$

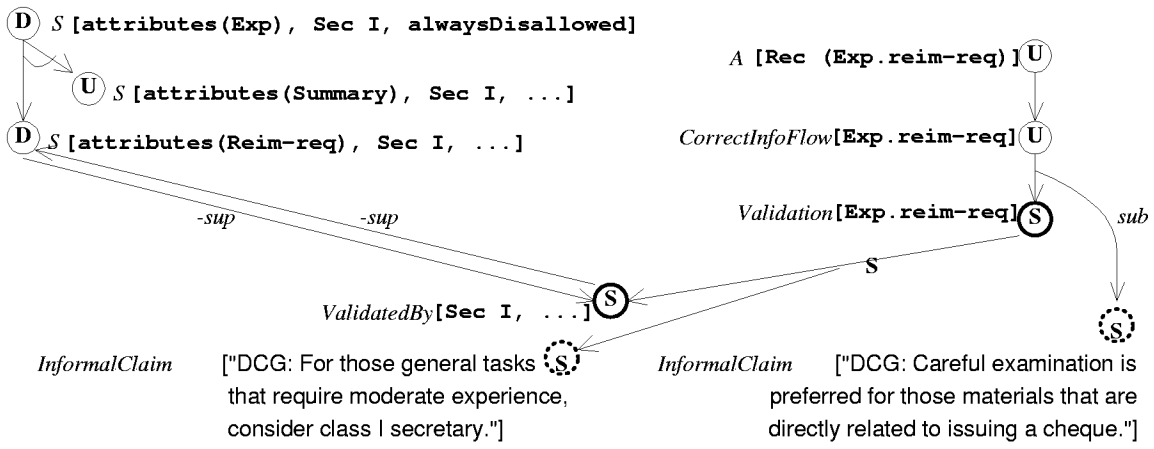


Figure 3.2: Mutual exclusion in satisficing accuracy of received reimbursement information.

Now suppose, as in Figure 3.2, that a security requirement was considered earlier: reimbursements should not be revealed to secretaries with a job classification below II. However, this is in direct conflict (i.e., mutually exclusive or sufficiently negative) with using a secretary of class I as the validator. Now the system uses the correlation rules to propose two new links with type *-sup*:

$$S[\texttt{attributes(Reim-req), SecI, AlwaysDisallowed}] \xrightarrow{-sup} \mathit{Validation}[\texttt{SecI, Exp.reim-req}, \ldots]$$

$$\mathit{Validation}[\texttt{SecI, Exp.reim-req}, \ldots] \xrightarrow{-sup} S[\texttt{attributes(Reim-req), SecI}, \ldots]$$

Suppose that the designer assigns $\mathit{FormalClaim}[\texttt{ValidatedBy[Sec I}, \ldots] \wedge \ldots]$ the label S (satisficed), and labels all the other leaves[8].

The labelling procedure of Section 2 propagates the labels upwards. Some of the results are shown in Figure 3.2. Since the validation by a class I secretary is sufficient counter-evidence, the security goal (See left-hand side of figure) is labelled D (denied). This value and the U value in the *AND* link in the upper-left corner are further propagated; the minimum value of the two is selected, resulting in D.

Note that the denial of the root security goal is not final. Instead of a class I secretary, the designer may see if a higher-ranking staff member can do the validation. Other satisficing methods may be considered as well. The designer will choose one alternative and provide an argument for later use in justifying the final design; then the labelling procedure will update the labels which reflect the current status of the process.

The success, or lack thereof, of goal satisficing methods relies on the cooperation between the system and agents in the environment, which is described in the *user's procedure manual,*[9] which is initially drafted during the design process. The manual indicates policies that the agents in the environment should obey when interacting with the system in order to satisfice the methods selected. For instance, if a *verification* method is selected, the manual indicates that a member must transfer his expense information to the system and to the project office which will enter the same information into the system.

At the design stage, the choice of methods (related to requirements for accuracy, security, and the like) results in selection among design alternatives.[10] In the next section, we consider how performance goals are dealt with in the implementation stage.

4 Dealing with Performance Requirements[11]

[8] To resolve conflicts, a negotiation-based approach may be taken (e.g., [40, 24]). We use argumentation methods to record how conflicts are resolved, e.g., by attachment of priorities.

[9] In acquiring formal requirements, [39] recognizes the need for generating documents which are in spirit similar to our manuals.

[10] See the description of dependency types in Section 2.3.

[11] An earlier version [36] of portions of this section appears in the *Proceedings of the Third International Workshop on Database Programming Languages,* Nafplion, Greece, August 1991.

The previous section illustrates the dynamic *process* aspect of design. This section focuses on *performance requirements,* as a second example of how a class of non-functional requirements can be treated within our proposed framework. Unlike accuracy requirements which were treated in the context of system design, performance requirements will be treated during the implementation phase when *designs* are mapped on to *implementations.*

A starting point for understanding good system performance is the set of standard definitions from computer systems theory (e.g., [27]), such as achieving low response time and suitable device utilizations. In practice,[12] performance goals often focus on response time and throughput, and are developed for particular application systems. They are often stated briefly, yet users expect the system to somehow meet their (implicit) performance concerns. And as we will see, performance goals can result in very complex goal-graph structures.

When implementing an information system using performance as a main criterion, the implementor has to abandon generic implementation algorithms and structures. Instead, implementation techniques have to be selected on a case-by-case basis from a number of alternatives. Inputs to this mapping process are: 1) a given set of *implementation alternatives*; 2) the *source schema* (some portion of the design specification); 3) a *workload characterization* for the particular system (e.g., an estimate of the number of researchers to be handled by the expense management system); 4) *performance goals,* specified for a particular system. As examples of performance goals, one could require that a researcher registering for a meeting should get from the system under design fast response time, and that storage requirements for information on all researchers be minimized. The framework detailed in section 2 is then applied for the satisficing of these qualitative goals. Outputs of the process are the target implementation, goal graphs, and a prediction of performance [36] calculated in terms of a performance model.

It is interesting to contrast the treatment offered in this section with other research based on the transformational approach, such as the TI system [2]. TI, like its transformation-based peers, focuses on correctness requirements, i.e., making sure that the generated implementation is consistent with the original specification. Performance, if treated at all, is treated as a selection criterion among alternative transformations. Kant's early work [25], on the other hand, does address performance goals. Her framework, however, focuses on conventional programming-in-the-small rather than information system development, relies on quantitative performance measures (which are available for her chosen domain but are, unfortunately, not available for information systems because of their complexity) and assumes an automatic programming setting rather than the dialectical software development process adopted here.

4.1 Layered Goal Structures

Since generating efficient implementations is better understood than some of the other phases of information system development, we can impose additional structure in the representation of performance goals. This is accomplished through a series of language layers, which account for potentially interacting data model features, implementation techniques and performance characteristics of design languages. This layered approach is inspired by a framework for prediction of performance of relational databases [22]. As design decisions are made at higher layers, corresponding to higher levels of abstraction, they are reflected in lower layers which describe the system in more detail. The layering shows where to introduce inputs related to design components, thus providing the information needed to make implementation decisions, while controlling the number of concepts to consider at a time.

We apply this layering approach to performance-based *selection* among implementation alternatives for conceptual design specification languages.[13] Our layering organizes some recent work on performance and implementation from the areas of semantic data models and object oriented systems.[14] For each layer, there are goal graphs whose refinements have an impact on graphs at lower layers. Figure 4.1 shows a series of linguistic subsets, where higher-level languages include more features supported by semantic data models: 0) The target *relational data model,* such as the database system facilities offered by the DBPL language [6]; 1) *Entities,* both persistent data entities (such as `John`, an instance of `Researcher`), and finite entities

[12] Many thanks to Michael Brodie for his insight on the use of performance goals in industry.

[13] See [36] for more on performance *prediction* for conceptual design specification languages.

[14] This includes results on record layout for entities and attributes [9, 46, 35, 4], enforcement of constraints [44, 8], and process scheduling [11].

(e.g., integers), arranged in *classes;* 2) *Attributes*, defined on entity classes, roughly corresponding to the Entity-Relationship Model [10]; 3) *Transactions,* modelled as classes with attributes and instance entities; 4) Entities and transactions with attributes, and classes arranged in *IsA hierarchies,* roughly corresponding to the Taxis subset described in [35]; 5) The above Taxis subset, extended with *constraints;* 6) The source conceptual design specification language, including constraints and *long-term processes* (whose nature has aspects of entities and activities, as well as constraints), comparable to Taxis [11] or TDL [6].

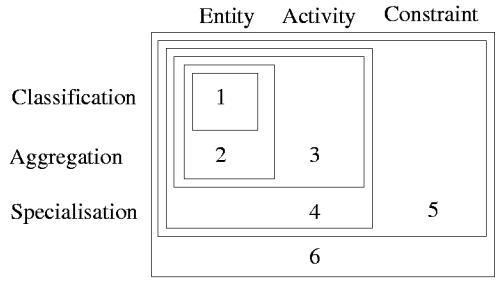


Figure 4.1: Layers arranged in a grid.

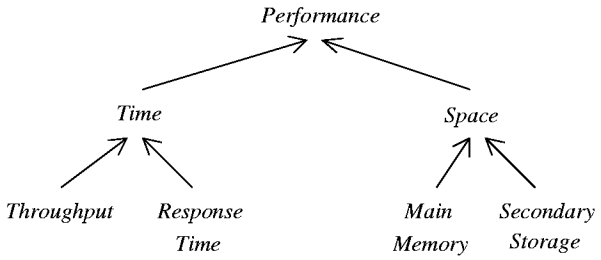


Figure 4.2: The Performance sort.

4.2 Performance Goal Refinement Methods

Performance goals drive selection of implementation alternatives, and are stated in terms of concepts applicable to information systems, such as response time. Many of our methods are based on features specific to information systems, and their implementation. All performance goals use the *Performance* sort. There are several *sub-sorts,* some of which are shown in Figure 4.2.

4.2.1 Goal Decomposition Methods

One aspect of goal decomposition involves the selection of an appropriate *sub-sort.* For example, we can use the *time-space* goal decomposition method to decompose the goal of "good performance for the Researcher class at layer 4" into the goals of good time performance for Researcher at layer 4 and good space performance for Researcher at layer 4:[15]

$$P[\texttt{Researcher, 4}] \xrightarrow{AND} \{Time[\texttt{Researcher, 4}],\ Space[\texttt{Researcher, 4}]\}$$

Likewise, a goal involving time can be decomposed by the *throughput-response time* method, and a goal involving space can be decomposed by the *main memory - secondary storage* method.

Another aspect of goal decomposition involves the decomposition of goal *parameters.* The *subclass* and *individualAttributes* performance goal decomposition methods are similar to the structural methods with the same names described in Section 3.

operational method. A performance goal on an information item `i` (such as a class, or an attribute of a class) can be decomposed into the corresponding goal for the *operations* $\texttt{o}_\texttt{j}$ on the item.

$$P[\texttt{i, Layer}] \xrightarrow{sub} \{P[\texttt{o}_\texttt{j}\texttt{(i), Layer}] \mid \texttt{o}_\texttt{j}\texttt{(i)}\ \textit{is an operation on}\ \texttt{i}\}$$

This method can be specialized. The *individual-bulk operations* method decomposes a goal on the basis of whether an operation manipulates one or many items. By the *implementation components* method, a goal for an operation is decomposed into lower-layer components of the operation.

static-dynamic schema method. While the conceptual design (or schema) of an information system may remain constant, in some cases it may be expected to change. For example, new specializations of `Researcher` might be added over time with relative efficiency, without requiring the entire system to be shut down and restarted. This method decomposes a performance goal for an information item, on the basis of whether the schema is expected to change.

[15] Here P stands for the *Performance* sort.

4.2.2 Goal Satisficing Methods

Some performance goal satisficing methods are available from systems performance engineering and semantic data model implementation techniques. *Indexing* is positive for time but negative for space. By *earlyFixing*, early connection is made between an action and the instructions that achieve it [43]. A specialization of *earlyFixing* is *staticOffsetDetermination*, which determines offsets statically, rather than at execution time. Using *accessManyAttributesPerTuple*, if many of the attributes in a tuple will frequently be accessed, time goals can be positively satisficed.

4.2.3 Goal Argumentation Methods

Expected or actual usage statistics, and predictions of performance of implementation alternatives, can be used as arguments for a choice of satisficing methods. Suppose we know that all references to information item `i` in a segment of code can be uniquely determined statically, rather than being expressions with several possible values. We write: *ExplicitReferences*`[i, Layer]`. An argument that information item is subject to frequent changes in the schema can be written: *FrequentSchemaChanges*`[i, Layer]`

4.3 Illustration

Returning to our research expense management system example, we will illustrate how a designer builds a goal graph for a few layers starting at Layer 4 (IsA hierarchies), showing some goal refinement methods and the impact of higher-layer goals upon lower ones. Figure 1.1 shows part of the IsA hierarchy for the example. Of the 12 attributes (not shown) of the `Researcher` class, ten, including `Name`, are inherited from `Employee`, while two others, including `Meeting`, are not inherited. Additional input information, such as the distribution of class populations, is required to characterize the workload. Of the 2000 employees, for instance, 1000 are researchers, including 700 computer researchers and 300 mathematicians. Of the two non-inherited attributes of `Researcher`, the `Meeting` attribute is very frequently accessed. This information can be included in argumentation structures.

Layer 4 selects implementations for attributes of entity classes; in the presence of IsA hierarchies, there are several possible implementation techniques. Inheritance hierarchies result in collections of attribute values whose appearance is more like a "staircase" than a relational table:[16]

	Name	Meeting	OperatingSystems
ComputerResearcher			
Researcher			
Employee			

As a result, a simple relational representation may waste space. Options include using one relation per class: storing either all attributes (newly defined or inherited) of a particular class in the corresponding relation (*horizontal splitting*), or only the newly defined attributes (*vertical splitting*).

Turning to the top of the goal graph (See Figure 4.3, and the legend for symbols in Figure 2.1.), the implementor's Layer 4 goal is good performance for the attributes of the `Researcher` entity class. First, the implementor decides to use the *time-space* method to decompose the goal into good time performance and good space performance for the attributes. The implementor can then use the *individual-bulk operations* method to decompose the time goal based on whether operations affect many entities, or just an individual entity. The goal of good time performance for individual operations on attributes of the `Researcher` class can now be decomposed by the *individualAttributes* method, resulting in goals for individual operations on the `Name` attribute, the `Meeting` attribute, etc. The implementor then focuses on the `Meeting` attribute, and observes that while most of the attributes of `Researcher` are inherited, `Meeting` is one of the two that is not. The implementor also recalls that `Meeting` is frequently accessed. By storing only the non-inherited attributes together, we have a small tuple size; moreover, of the attributes which are stored in the tuple, a high proportion will be frequently accessed. The actual value of this ratio (50%) is recorded as an argument for selecting the satisficing goal of improving time performance by accessing many attributes per tuple. This satisficing goal leads to selection of an implementation using vertical splitting for the attributes of

[16] In the illustration, not all attributes are shown.

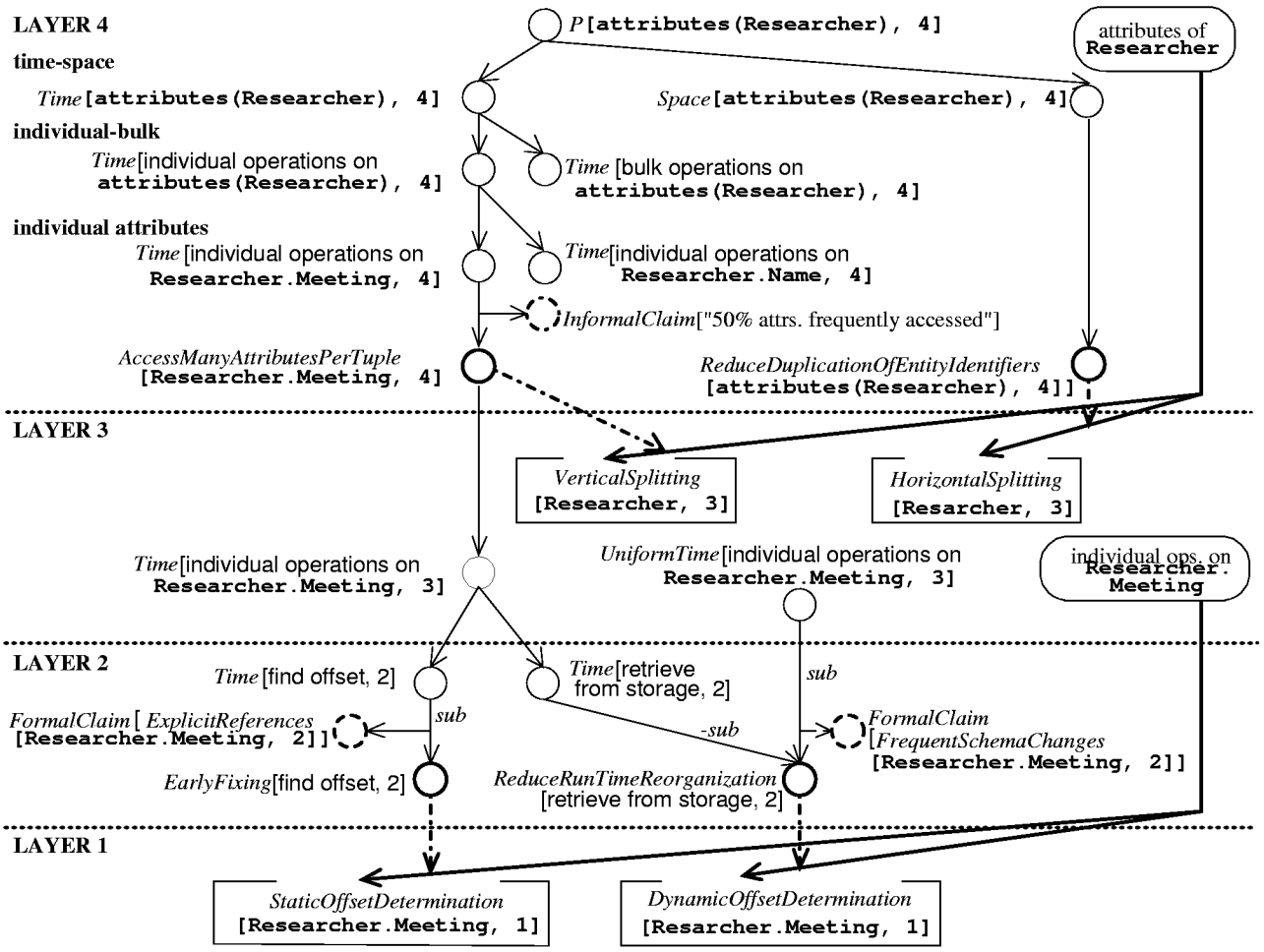


Figure 4.3: A Performance goal graph.

`Researcher` and its specializations. Another implementation alternative is horizontal splitting, which can offer better space performance.

At Layer 4, the designer dealt with the `Researcher` class in an IsA hierarchy, leaving the mapping target at Layer 3 being the `Researcher` class using vertical splitting. The satisficing goal *AccessManyAttributesPerTuple*`[...]` is refined to the Layer 3 (transactions) goal of good time performance for individual operations on the `Meeting` attribute of `Researcher`. Thus the implementor continues addressing the goal of good time performance, but at Layer 3, which deals with operations without inheritance.

The implementor decomposes the Layer 3 *Time* goal (See middle left of Figure 4.3) according the implementation components of the operation (only some of which are shown). The result is a set of Layer 2 (attributes) time goals — for finding the offset for the `Meeting` attribute field within a relational tuple, retrieving the value from secondary storage, etc. The implementor focuses on finding the offset quickly; *earlyFixing* is positive. The implementor reviews the source schema and observes that `Meeting` is always referenced explicitly in the code. *ExplicitReferences*`[Researcher.Meeting, 2]` is recorded as an argument for the *sub* link, and static offset determination for the `Meeting` attribute is chosen as an implementation technique. Thus the implementor has dealt with a Layer 2 issue, resulting in a mapping target at Layer 1.

An alternative implementation is dynamic offset determination. The *-sub* link records its negative impact on the goal of minimizing time. However, this would have a positive impact on another goal — offering *uniform* time performance. As shown in the lower right-hand side of Figure 4.3, when dealing with frequent schema changes, a structure which reduces expensive run-time reorganization can offer less variation in

response time.

5 Conclusions

The main contribution of this research is that it offers a concrete framework for integrating non-functional requirements into the software development process, at least for information systems. In tackling this task, our research extends earlier work by Lee [28, 29] and [38, 14]. The framework is still under refinement and a prototype implementation is under way, intended to provide a vehicle for more thorough testing and for gaining experience with the framework's strengths and weaknesses.

Much remains to be done with this work. Firstly, the framework needs to be applied to other types of non-functional requirements and life-size examples. Secondly, the framework needs a theoretical foundation for representing and reasoning with non-functional requirements. This foundation needs to include a semantics for non-functional requirements. For example, what does it really mean to claim that a particular design decision enhances system accuracy concerning employee data? Moreover, a proof theory based on this semantics is required, including efficient algorithms for special classes of inferences related to non-functional requirements. The whole framework we have offered here can then be justified on formal semantic grounds rather than informal, intuitive ones.

Unfortunately, it seems that such a formal semantic treatment of non-functional requirements would need to be done individually for different types of requirements and is therefore a long term research project. In the meantime, an experimental approach such as the one adopted here can offer solutions that may find immediate use in an area of computer practice that is in great need of concepts, methodologies and tools.

Acknowledgments

We would like to thank the referees for their constructive and detailed comments, as well as Eric Yu and Sheila McIlraith for providing helpful suggestions.

References

[1] *Artif. Intell. J.,* vol. 24, nos. 1–3, Dec., 1984.

[2] R. Balzer, "A 15 Year Perspective on Automatic Programming" *IEEE Trans. Software Eng.,* vol. SE–11, no. 11, Nov. 1985, pp. 1257–1268.

[3] V. R. Basili and J. D. Musa, "The Future Engineering of Software: A Management Perspective," *IEEE Computer,* vol 24, no. 9, Sept. 1991, pp. 90–96.

[4] V. Benzaken, "An Evaluation Model for Clustering Strategies in the O_2 Object-Oriented Database System," in *Proc. 3rd Int. Conf. Database Theory,* Paris, Dec. 1990. Berlin: Springer-Verlag, 1990, pp. 126–140.

[5] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M. J. Merritt, *Characteristics of Software Quality.* Amsterdam: North-Holland, 1978.

[6] A. Borgida, J. Mylopoulos, J. W. Schmidt and I. Wetzel, "Support for Data-Intensive Applications: Conceptual Design and Software Development," in *Proc. 2nd Int. Workshop on Database Programming Languages,* June 1989, Gleneden Beach, Oregon. San Mateo, CA: Morgan Kaufmann, 1990, pp. 258–280.

[7] T. P. Bowen, G. B. Wigle and J. T. Tsai, "Specification of Software Quality Attributes," Rep. RADC–TR–85–37, Rome Air Development Center, Griffiss Air Force Base, NY, Feb. 1985.

[8] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Management," in *Proc. 16th Int. Conf. Very Large Data Bases,* Brisbane, Australia, Aug. 1990, pp. 566–577.

[9] A. Chan, S. Danberg, S. Fox, W.-T. K. Lin, A. Nori and D. Ries, "Storage and Access Structures to Support a Semantic Data Model," in *Proc. 8th Int. Conf. Very Large Data Bases,* Mexico City, Sept. 1982, pp. 122–130.

[10] P. P.-S. Chen, The Entity-Relationship Model — Toward a Unified View of Data. *ACM Trans. Database Systems,* vol. 1, no. 1, March 1976, pp. 9–36.

[11] K. L. Chung, D. Rios-Zertuche, B. A. Nixon and J. Mylopoulos, "Process Management and Assertion Enforcement for a Semantic Data Model," in *Proc. EDBT '88, Int. Conf. Extending Database Technology,* Venice, Italy, March 1988. Berlin: Springer-Verlag, 1988, pp. 469–487.

[12] L. Chung, "Representation and Utilization of Non-Functional Requirements for Information System Design," in *Proc. CAiSE '91, 3rd Int. Conf. Advanced Information Systems Eng.,* Trondheim, Norway, May 1991. Berlin: Springer-Verlag, 1991, pp. 5–30.

[13] K. L. Chung, P. Katalagarianos, M. Marakakis, M. Mertikas, J. Mylopoulos and Y. Vassiliou, "From Information System Requirements to Designs: A Mapping Framework," *Information Systems,* vol. 16, no. 4, 1991, pp. 429–461.

[14] J. Conklin and M. L. Begeman, "gIBIS: A Hypertext Tool for Explanatory Policy Discussions," *ACM Trans. Office Information Systems*, vol. 6, no. 4, 1988, pp. 303–331.

[15] J. de Kleer, "Problem Solving with the ATMS," *Artif. Intell. J.,* vol. 28, 1986, pp. 127–162.

[16] C. DiMarco, "Computational Stylistics for Natural Language Translation," Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1990.

[17] J. Doyle, "A Truth Maintenance System," *Artif. Intell. J.,* vol. 12, 1979, pp. 231–272.

[18] S. F. Fickas, "Automating the Transformational Development of Software," *IEEE Trans. Software Eng.,* vol. SE–11, no. 11, Nov. 1985, pp. 1268–1277.

[19] U. Hahn, M. Jarke and T. Rose, "Teamwork Support in a Knowledge-Based Information Systems Environment," *IEEE Trans. Software Eng.,* vol. 17, no. 5, May 1991, pp. 467–482,

[20] H. R. Hartson and D. K. Hsiao, "Full Protection Specifications in the Semantic Model for Database Protection Languages," in *Proc. ACM Annual Conf.*, Houston, TX, Oct. 1976, pp. 90–95.

[21] J. R. Hauser and D. Clausing, "The House of Quality," *Harvard Business Review,* May-June 1988, pp. 63–73.

[22] W. F. Hyslop, "Performance Prediction of Relational Database Management Systems," Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1991.

[23] M. Jarke, J. Mylopoulos, J. W. Schmidt and Y. Vassiliou, "DAIDA: An Environment for Evolving Information Systems," *ACM Trans. Information Systems*, vol. 10, no. 1, Jan. 1992, forthcoming.

[24] W. L. Johnson, M. S. Feather, D. R. Harris and K. M. Benner, "Representation and Presentation of Requirements Knowledge." Manuscript, USC/Information Sciences Institute, Oct. 1991.

[25] E. Kant, "On the Efficient Synthesis of Efficient Programs," *Artif. Intell. J.,* vol. 20, no. 3, May 1983, pp. 253–305.

[26] S. E. Keller, L. G. Kahn and R. B. Panara, "Specifying Software Quality Requirements with Metrics," in *Tutorial: System and Software Requirements Engineering*, R. H. Thayer and M. Dorfman, Eds. IEEE Computer Society Press, 1990, pp. 145–163,

[27] E. D. Lazowska, J. Zahorjan, G. S. Graham and K. C. Sevcik, *Quantitative System Performance.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

[28] J. Lee, "SIBYL: A Qualitative Decision Management System," in *Artificial Intelligence at MIT: Expanding Frontiers*, vol. 1, P. H. Winston and S. A. Shellard, Eds. Cambridge, MA: The MIT Press, 1990, pp. 105–133.

[29] J. Lee, "Extending the Potts and Bruns Model for Recording Design Rationale," in *Proc. 13th Int. Conf. Software Eng.*, Austin, TX, May 1991, pp. 114–125.

[30] J. Martin, *Security, Accuracy, and Privacy in Computer Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[31] J. Mostow, "Towards Better Models of the Design Process," *AI Magazine*, vol. 6, no. 1, Spring 1985, pp. 44–57.

[32] J. Mylopoulos, P. A. Bernstein and H. K. T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. Database Systems,* vol. 5, no. 2, June 1980, pp. 185–207.

[33] J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis, "Telos: Representing Knowledge about Information Systems," *ACM Trans. Information Systems,* vol. 8, no. 4, Oct. 1992, pp. 325–362.

[34] N. Nilsson, *Problem-Solving Methods in Artificial Intelligence.* New York, McGraw-Hill, 1971.

[35] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos and M. Stanley, "Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis," in *Proc. ACM SIGMOD 1987 Annual Conf.,* San Francisco, CA, May 1987, (ACM SIGMOD Record, vol. 16, no. 3, Dec. 1987), pp. 118–131.

[36] B. Nixon, "Implementation of Information System Design Specifications: A Performance Perspective," in *Proc. 3rd Int. Workshop on Database Programming Languages,* Nafplion, Greece, Aug. 1991, P. Kanellakis and J. W. Schmidt, Eds. San Mateo, CA: Morgan Kaufmann, forthcoming.

[37] C. P. Pfleeger, *Security in Computing.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[38] C. Potts and G. Bruns, "Recording the Reasons for Design Decisions," in *Proc. 10th Int. Conf. Software Eng.*, 1988, pp. 418–427.

[39] H. Reubenstein, "Automated Acquisition of Evolving Informal Descriptions," Ph.D. Thesis; also Tech. Rep. 1205, MIT Artif. Intell. Lab., 1990.

[40] W. N. Robinson, "Negotiation Behavior During Requirement Specification," in *Proc. 12th Int. Conf. Software Eng.,* Nice, France, March 1990, pp. 268–276.

[41] G.-C. Roman, "A Taxonomy of Current Issues in Requirements Engineering," *IEEE Computer*, vol. 18, no. 4, Apr. 1985, pp. 14–23.

[42] H. A. Simon, *The Sciences of the Artificial,* 2nd ed. Cambridge, MA: The MIT Press, 1981.

[43] C. U. Smith, *Performance Engineering of Software Systems.* Reading, MA: Addison-Wesley, 1990.

[44] M. Stonebraker, "Triggers and Inference in Database Systems," in *On Knowledge Base Management Systems,* M. L. Brodie and J. Mylopoulos, Eds. New York: Springer-Verlag, 1986, pp. 297–314.

[45] R. H. Thayer and M. C. Thayer, "Glossary," in *Tutorial: System and Software Requirements Engineering*, Richard H. Thayer and Merlin Dorfman, Eds. IEEE Computer Society Press, 1990, pp. 605–676.

[46] G. E. Weddell, "Selection of Indexes to Memory-Resident Entities for Semantic Data Models," *IEEE Trans. Knowledge and Data Eng.,* vol. 1, no. 2, June 1989, pp. 274–284.

Contents