4

METHODS AND CORRELATION RULES

During the process of software development, the developer uses *softgoals* and *contributions* to analyze and record in a softgoal contribution graph her intentions, design alternatives and tradeoffs and rationale, and uses the *evaluation procedure* to evaluate where her intentions are properly addressed.

In this chapter, we present the remaining components: *methods* and *correlation rules*. These two components help the developer generate a softgoal contribution graph, by allowing for knowledge about ways of softgoal refinements and their interactions to be captured, encoded, catalogued, tailored, reused and improved.

4.1 METHODS

With the three types of softgoals and several contribution types, the developer now can represent NFRs and iteratively refine them into other more specific NFRs. When NFRs are sufficiently refined, the developer then introduces development techniques to satisfice such NFRs. For each refinement, design decisions can be recorded in terms of arguments.

During this process, softgoals may be refined *manually* by the developer, who is then responsible for satisficing not only the softgoal's offspring but also the refinement itself represented as a contribution.

However, a manual refinement is often time-consuming especially when the knowhow for such a refinement is not readily available, as it takes time and effort to search for the source of the knowhow. Even when available, the knowledge needed for making a manual refinement is not properly encoded, hence not easily to be shared, extended, tailored, or reused.

In order to alleviate these difficulties, the framework provides *refinement methods* (*methods* in short) which represent generic procedures for refining a softgoal or contribution into one or more offspring. When a number of methods are collected and catalogued, they can offer appropriate vocabulary and subject matter in dealing with NFRs.

For example,

> "To maintain accurately data about accounts, you need to maintain accurately data about regular accounts, gold accounts and premier accounts."

might come from experts in the credit card system domain (or a similar one, e.g., a banking system domain). Once acquired, this knowhow might be represented as a method:

Method maximize-account-accuracy
    parent: Accuracy [Account]
    offspring: {Accuracy [RegularAccount],
        Accuracy [GoldAccount],
        Accuracy [PremierAccount]}
    contribution type: AND

Here maximize-account-accuracy is the name of a method which, given a parent goal Accuracy[Account], generates three offspring goals Accuracy[RegularAccount], Accuracy[GoldAccount], and Accuracy[PremierAccount]. Since the contribution type is AND, the parent will be satisficed if all the offspring are.

Figure 4.1 (a) pictorially depicts this method definition. In the pictorial representation, each softgoal is shaded in order to indicate that it is part of a method definition, *not* of a method application which introduces new (unshaded) softgoals in a softgoal contribution graph.

Once a method is defined, it can be used to (automatically) generate parts of a softgoal contribution graph. Given a particular softgoal or contribu-

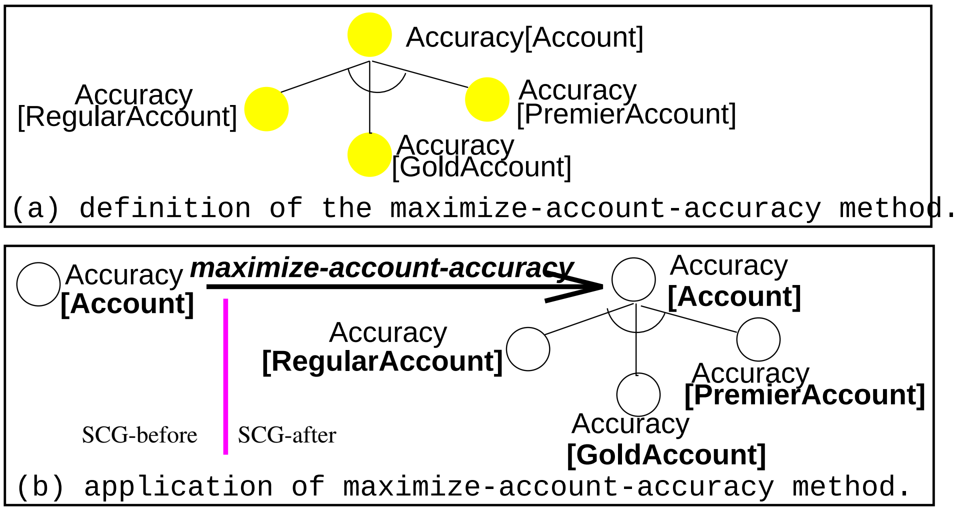

Figure 4.1 The maximize-account-accuracy method and an application.

tion in a softgoal contribution graph, its offspring are generated according to the definition of the method being applied.

For example, the method maximize-account-accuracy can be applied to Accuracy[Account], which results in the generation of three offspring goals and a contribution (here, AND):

MethodApplication maximize-account-accuracy
    parent: Accuracy [Account]
    offspring: {Accuracy [RegularAccount],
        Accuracy [GoldAccount],
        Accuracy [PremierAccount]}
    contribution type: AND

This method application is pictorially represented in Figure 4.1 (b).

Note that a softgoal contribution graph does not show this kind of figure in its entirety, but only the right-hand side which represents the already-existing parent, its newly-created offspring and their contribution towards the satisficing of the parent.

In order to be sure about this, let us see how a method application works. Suppose that the developer has selected Accuracy[Account] as the goal to refine. Also suppose that she has selected maximize-account-accuracy as the method to apply to the goal. At this point, the chosen method is applicable to the chosen goal only if both the sort in the method definition matches that of the goal and the parameter list in the method definition matches that of the goal. If the method is indeed applicable, offspring of the goal are generated, along with a contribution, as specified in the method definition.

In the case of maximize-account-accuracy, a method takes a parent having a fixed sort (here, Accuracy) and a fixed parameter list (here, Account), and then generate offspring whose sorts and parameter lists are again fixed.

While this kind of method is useful for capturing domain-specific refinement knowledge, a method becomes less domain-dependent and more amenable to tool support when the sort and parameter list expression involves variables.

For example, the following is a method which takes a variable as parameter:

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

With a variable in the parameter, the above knowhow might be represented as follows (variables are italicized):

DecompositionMethod accuracy-subclass
- parent: Accuracy [x]
- offspring: {Accuracy [y_1], Accuracy [y_2], $\cdots$, Accuracy [y_n]}
- contribution type: AND
- applicabilityCondition: /* x must be an information class */
- constraint: i y_i isA x and
  - /* set up one offspring for every subclass of the parent parameter */

Unlike the definition of the maximize-account-accuracy earlier, this definition starts with DecompositionMethod instead of Method. As will be described shortly, there are three major types of methods, namely, decomposition methods (DecompositionMethod), operationalization methods (OperationalizationMethod) and argumentation methods (ArgumentationMethod), corresponding closely to the three types of softgoals introduced earlier. The keyword Method can be replaced by one of the three method type designators, according to the type of the method currently being defined.

As seen in the definitions of both maximize-account-accuracy and accuracy-subclass-method, the body of each method definition specifies the sort and parameter(-list) of the parent softgoal or contribution followed by a list of offspring sort and offspring parameter(-list) pairs. As before, *contribution-type* is the contribution type relating the offspring to the parent.

applicabilityCondition describes the kind of parent in a softgoal contribution graph to which the method can be applied. Here the parameter of the parent parameter must be an information class.

In contrast, constraint describes what kind of offspring should be generated when the method is applied to the parent. Here, one offspring should be generated for every subclass of the parent parameter.

Note the convention that all free variables are universally quantified. In other words, "i y_i isA x" is equivalent to "forall i y_i isA x".

As seen in the body of the above method definition, both the applicabilityCondition and constraint can be described either formally, informally or both.

Figure 4.2 (a) pictorially depicts this method. As seen in this pictorial notation, only the sorts and parameters may appear, while the remaining applicabilityCondition and constraint are usually omitted.

When the parameter description in the body of a method involves a variable, an application of the method can benefit from the functional requirements description. Among other things, the description usually includes *structural requirements* which model information structures and rules which determine allowable states of these structures, and *behavioral requirements* which model activities that operate on such structures and events that trigger these activities.

Suppose that, in the structural requirements description, the class of accounts has three subclasses, namely, the class of regular accounts, the class of gold accounts and the class of premier accounts. This specialization hierarchy

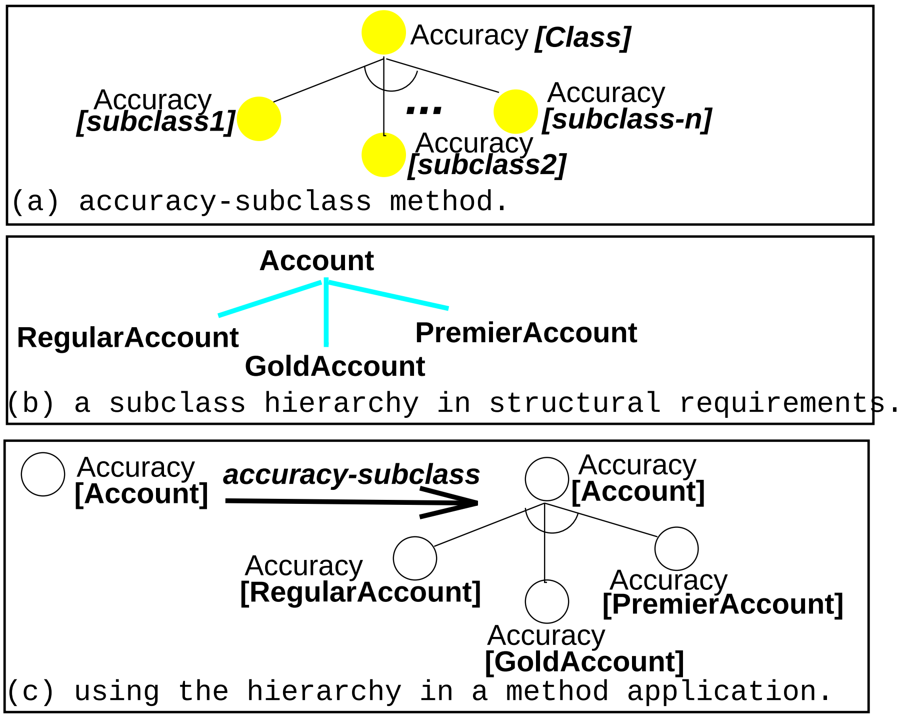

Figure 4.2 The accuracy-subclass method and its application.

is shown in Figure 4.2 (b). Then the application of the accuracy-subclass method to Accuracy[Account] will result in the generation of three offspring goals. Figure 4.2 (c) illustrates the method application.

Figure 4.2 as a whole, thus, illustrates the definition of a method, where the parameter is variable, the definition of a class hierarchy in the structural requirements description, and finally how an application of the method works according to the class hierarchy.

Note that the net result of all this is identical with the application of the earlier maximize-account-accuracy method. Why do we bother with the accuracy-subclass method, then, since the use of the maximize-account-accuracy method does not even need a class hierarchy?

It's because the maximize-account-accuracy cannot be applied to any softgoal but Accuracy[Account]. In contrast, for the accuracy-subclass method, the parameter can be just about anything pertaining to information. In other words, the method can be applied not only to the accuracy of all account information but also to the accuracy of all medical diagnosis information, which should be of concern to patients and practitioners alike in the context of health care system. Now if each diagnosis falls into one of psychiatric, alcoholic, cancer, and meningitis diagnoses, then the method application results in the generation of four offspring, the parameters of each being one of the four types. Figure 4.3 illustrates this. Part (a) is as before; parts (b) and (c) respectively

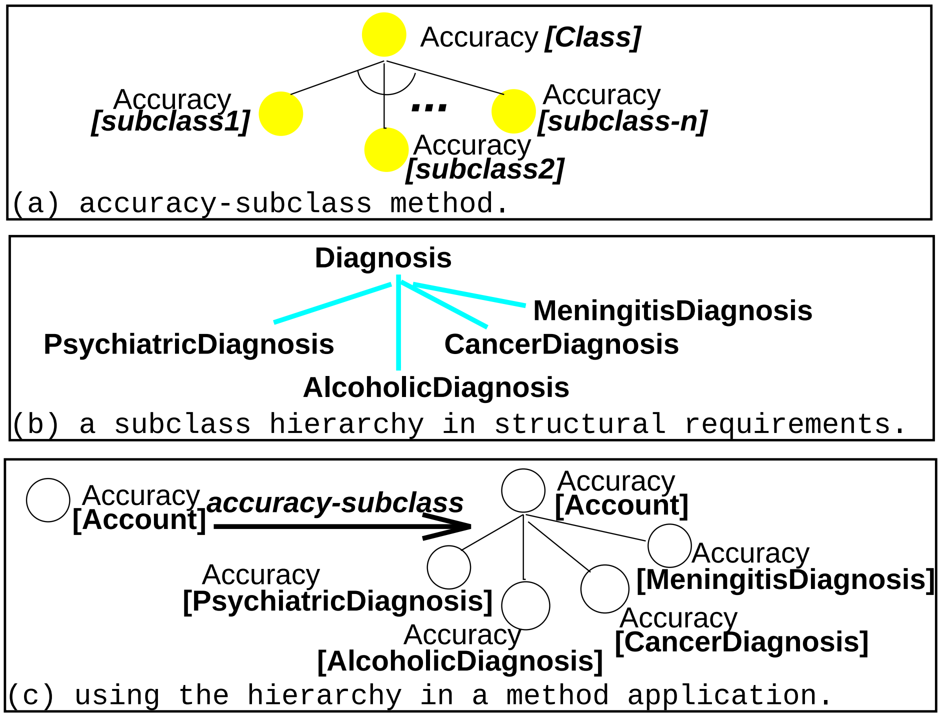

Figure 4.3 The accuracy-subclass method applied to "Information about diagnosis must be accurate".

show a new class hierarchy for the parameter and a new application of the method.

Softgoals with multiple parameters are refined just the way softgoals with a single parameter are, as illustrated below for an operating cost goal with two parameters:

Method minimize-manpower-operating-cost

parent: OperatingCost [manpower, $100 000]

offspring: {OperatingCost [full-time-employee, $100 000],
OperatingCost [part-time-employee, $100 000]}.

contribution type: AND

Just as the maximize-account-accuracy method was generalized to the accuracy-subclass method, the minimize-operating-cost can be generalized:

> "To minimize the operating cost for type **x** with the upper limit **c**, you need to minimize the operating cost for the components of **x**, each with the same upper limit."

This knowhow might be represented as follows (variables are again italicized):

Method minimize-operating-cost

parent: OperatingCost [x, c]

offspring: {OperatingCost [y_1, c], OperatingCost [y_2, c], $\cdots$,
OperatingCost [y_n, c]}

contribution type: AND

applicabilityCondition: /* x must be a type of operating cost
and y a monetary amount */

constraint: i y_i componentOf x
/* set up one offspring for every component
of the parent parameter */

An application of both the minimize-manpower-operating-cost and the minimize-operating-cost methods then results in the generation of two offspring goals, each again with two parameters. Figure 4.4 pictorially depicts an application of the minimize-operating-cost.

Now onto the three major types of methods mentioned earlier, namely, decomposition, operationalization, and argumentation methods. Let us start with decomposition methods.

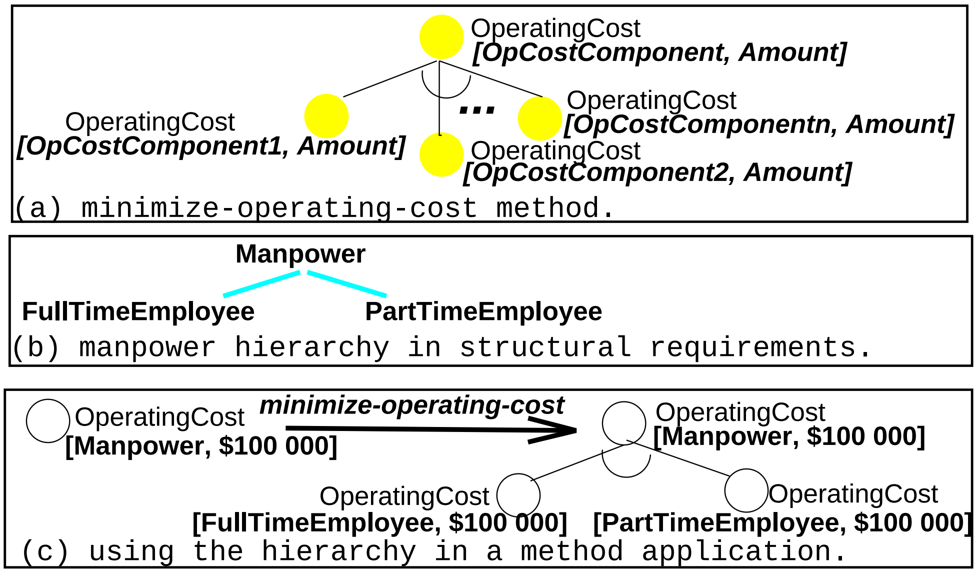

Figure 4.4 The accuracy-subclass method and its application.

4.2 DECOMPOSITION METHODS

Initial NFR goals are often ambiguous and invite many interpretations from different groups of people. They are also coarse-grained and do not permit the consideration of design decisions which normally require more specific details. Similarly, both operationalized goals and claims can also be unclear and coarse-grained.

Decomposition methods are for capturing, encoding and compiling into a catalogue the knowhow specifically for different (and possibly conflicting) ways of interpreting softgoals and breaking them down into more detailed ones. Decomposition methods, thus, allow for multiple conflicting interpretations of non-functional requirements to coexist as a natural phenomenon, and enable the developer to choose the one that best suits the needs of the intended application domain.

Decomposition methods also allow for the "divide-and-conquer" paradigm to be exercised. This helps deal with as many NFRs as needed, perhaps in parallel by multiple teams.

Of course, as with other types of methods, a catalogue of decomposition methods can be shared and reused, within the same and across different application contexts, hence alleviating the time-consuming, and often difficult, search for the knowhow.

Decomposition methods are usually an AND decomposition of a parent softgoal or contributory relationship into a group of offspring.

Two important types of decomposition methods are *sort decomposition methods* and *parameter decomposition methods*.

4.2.1 Sort Decomposition Methods

Sort decomposition methods are specializations of decomposition method which are in turn specializations of methods. As such, sort decomposition methods inherit all the benefits of decomposition methods and methods in general. Additionally, these methods allow specifically for relating and correlating sorts of NFR goals which represent the types of system characteristics, such as accuracy, operating cost, modifiability and traceability.

These methods also allow for relating and correlating sorts of operationalized goals which represent the types of development techniques in the target design or implementation which help achieve non-functional requirements as goals.

Unlike those for NFR goals and operationalized goals, sort decomposition methods for claims can relate only between and across Claim and Argument.

Regardless of the types of softgoals, a sort decomposition method allows for the refinement of a parent on its sort in terms of offspring, each with a subsort of the parent sort. Each subsort can be viewed as representing special cases for each softgoal class.

For instance, as described in Chapter 2, the Security sort may have subsorts Integrity, Confidentiality, and Availability. Using this relationship, a sort decomposition method can be encoded as follows:

DecompositionMethod maximize-security-of-account
    parent: Security [Account]

offspring: {Integrity [Account],
Confidentiality [Account],
Availability [Account]}
contribution type: AND

Figure 4.5 (a) and Figure 4.5 (b) respectively depict the definition and an application of this method. As before, each softgoal is again shaded in order

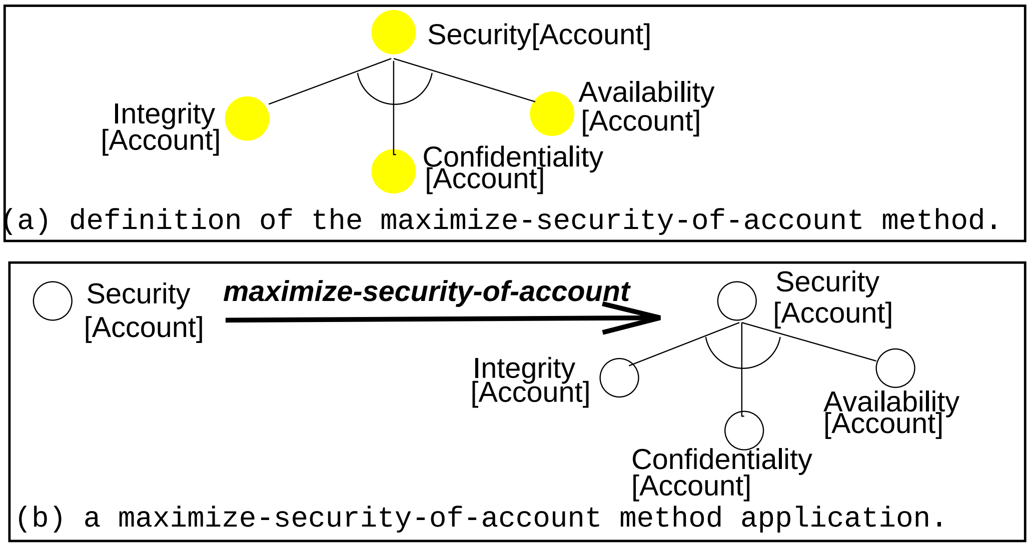

Figure 4.5 The maximize-account-accuracy method and its application.

to indicate that it is part of a method definition, *not* of a method application which introduces new (unshaded) softgoals in a softgoal contribution graph.

Sort decomposition methods also allow for variables to be used in place of the sort, yielding the following general form:

DecompositionMethod subsort-with-account
parent: s [Account]
offspring: {s_1[Account], s_2 [Account], $\cdots$, s_n [Account]}
contribution type: AND
applicabilityCondition: /* s must be a sort */

constraint: i s_i isA s and

/* set up one offspring for every subclass of the parent sort */

Figure 4.6 (a) and Figure 4.6 (b) respectively depict this method definition and an application of it. Note that the parameter of the parent is copied

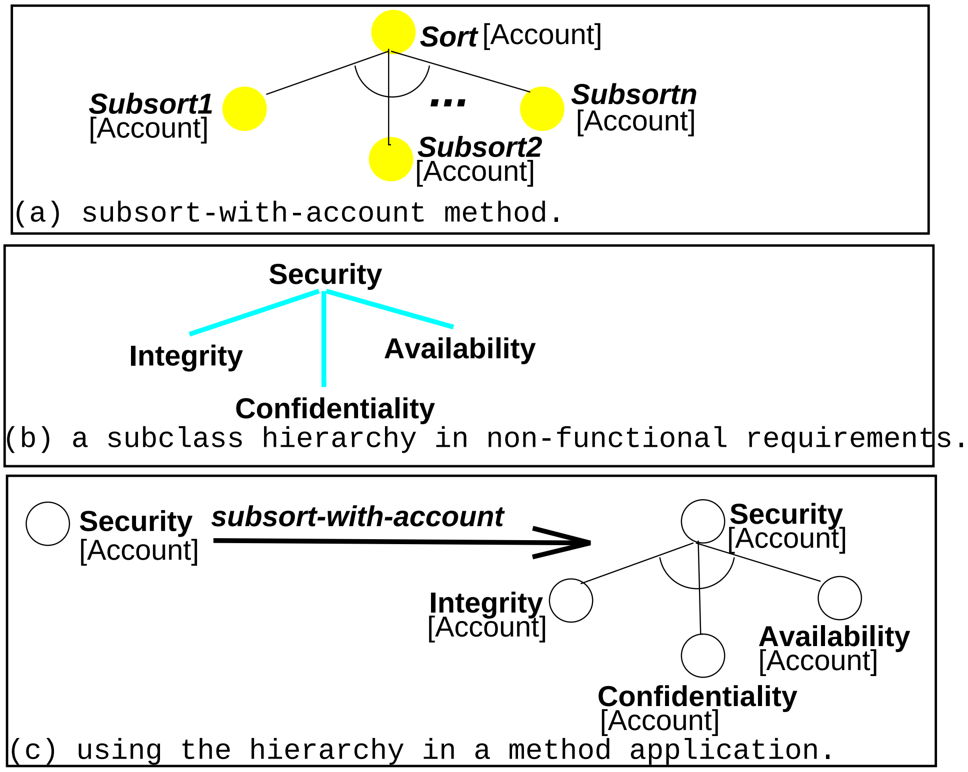

Figure 4.6 The subsort-with-account method and one of its applications.

down as that of the offspring; only the sorts of the offspring differ from that of the parent.

Of course, the subsort-with-account method can take any goal as the parent, as long as the goal has the class of accounts as its parameter. For example, the method can be applied to Adaptability [Account] and result in the generation of ChangeDetectability [Account] and Modifiability [Account], assuming that the sort Adaptability has two subsorts ChangeDetectability and Modifiability. Figure 4.7 depicts the use of variable sort pictorially.

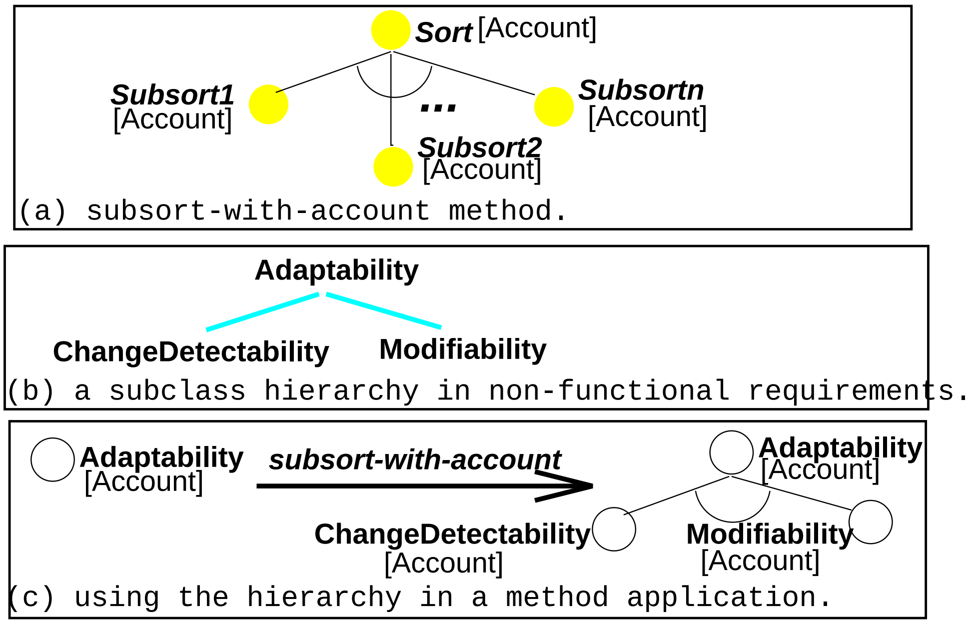

Figure 4.7 The subsort-with-account method and one of its applications.

Thus, the subsort-with-account method may be viewed as being more generic than the maximize-security-of-account method. An even more generic method can be defined which can be applied to virtually any softgoal having variables for both the sort and parameters:

DecompositionMethod subsort
- parent: s [p]
- offspring: $\{s_1 [p], s_2 [p], \cdots, s_n [p]\}$
- contribution type: AND
- applicabilityCondition: /* s must be a sort */
- constraint: i s_i isA s and
  /* set up one offspring for every subclass of the parent sort */

Now, this method can be applied not only to Security[Account] and Adaptability[Account], but Security[Customer], Security[Address], Adaptability[Customer], Adaptability[Address], etc. This method is quite domain-independent, and as such it can be used in any context: systems for banking, credit card, insurance, health, military, etc.

To clarify this domain independence, let us consider another application of the method to, say a performance goal such as Performance[InsuranceClaimsProcessing]. Now the sort Performance typically is associated with two subsorts Time and Space, representing respectively time and space performance requirements on a particular system. Assuming this sort-subsort has already been established, Figure~\ref{Insurance} illustrates the application of the subsort-method to the performance goal.

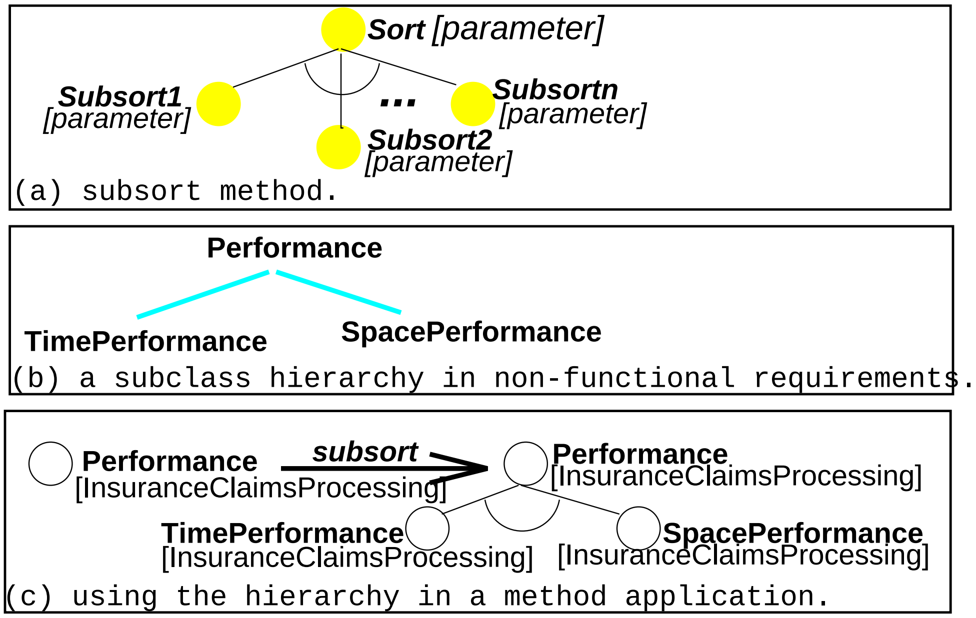

Figure 4.8 The subsort method and one of its applications.

Figure 4.9 depict the three sort decomposition methods with increasing degree of genericity: maximize-security-of-account, subsort-with-account and subsort methods.

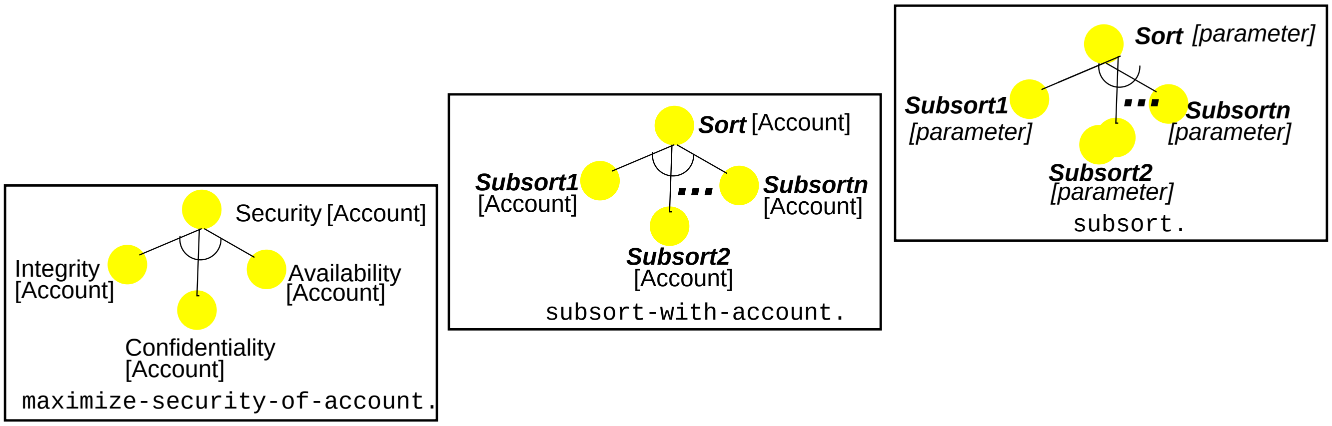

Figure 4.9 Three sort decomposition methods with increasing degree of genericity.

As with non-functional requirements, operationalized goals and claims too can sometimes be ambiguous and need to be more detailed. In this case, they also need to be refined through decomposition methods. As a matter of fact, an example of such a refinement was given earlier in Chapter 2 for authorization as a technique to address confidentiality of accounts. This refinement could have been based on the following method:

DecompositionMethod refine-authorization
    parent: Authorization [*Information*]
    offspring: {Identification[*Information*],
        Authentication [*Information*],
        AccessRuleValidation [*Information*]}
    contribution type: AND

Figure 4.10 pictorially depicts the definition of this method and one of its applications. Note that operationalized goals are denoted by thick circles.

4.2.2 Parameter Decomposition Methods

Like sort decomposition methods, Parameter decomposition methods are specializations of decomposition methods which are in turn specializations of

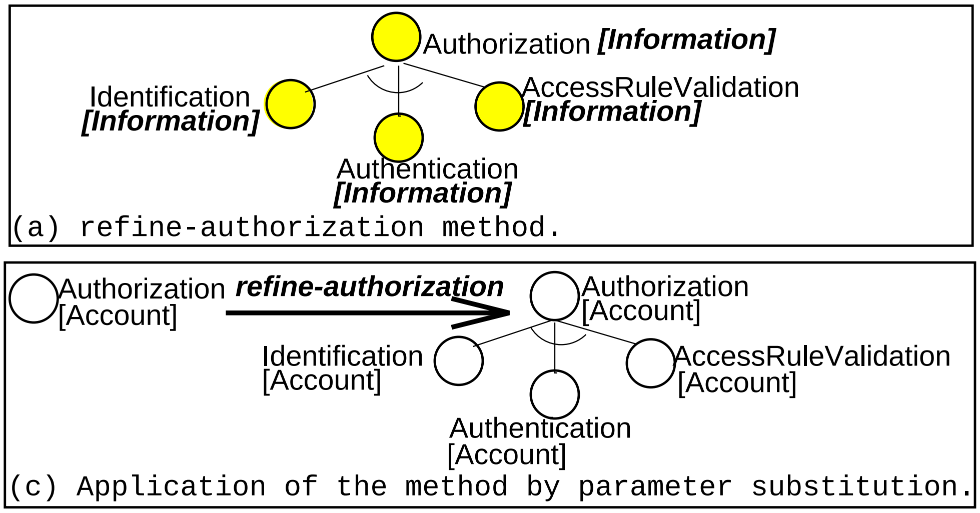

Figure 4.10 The refine-authorization method and one of its applications.

methods. As such, parameter decomposition methods inherit all the benefits of decomposition methods and methods in general. Additionally, these methods allow specifically for relating parameters of softgoals and contributions to each other.

A parameter of a softgoal can refer to a collection of items, such as information, processes, functions, events, interfaces, time, monetary amount, etc. Used for describing functional aspects of the system, these items are usually organized along structural modelling primitives of functional requirements, system and software architectures, or process architectures. Such primitives include classes, super-/sub-classes, attributes, views, contexts, perspectives, packages/components, compartments, etc.

Whatever a parameter may refer to, it can be decomposed along a structural dimension. For instance, the minimize-operating-cost mentioned earlier is a parameter decomposition method which decomposes [manpower, $100 000] into [full-time-employee, $100 000] and [part-time-employee, $100 000].

The aforementioned maximize-account-accuracy and more generic accuracy-subclass methods are also parameter decomposition methods which decompose the parameter of an accuracy goal in terms of subclasses of the parameter of the goal.

Recall that the subsort method is more generic than the subsort-with-account method which, in turn, is more generic than the maximize-security-of-account method.

Similarly, a parameter decomposition method can be defined which is even more generic than the accuracy-subclass method, and can be applied to virtually any softgoal having variables for both the sort and parameters. The subclass method presented below decomposes any softgoal or contribution having a class as the parameter into softgoals having as parameters its immediate specializations:

DecompositionMethod subclass
    parent: s $[p]$
    offspring: $\{s\ [p_1],\ s[p_2],\ \cdots,\ s[p_n]\}$
    contribution type: AND
    applicabilityCondition: /* s must be a sort and p a class */
    constraint: i p_i isA p and
        /* set up one offspring for every subclass
        of the parent class */

Now with the subclass-method, the same result can be obtained as with the more specialized accuracy-subclass-method: Figure 4.11 (a) and Figure 4.11 (b) respectively depicts the definition of this method and one of its applications. pictorially depicts this method. Note that the sort of the parent is copied down as that of the offspring; only the parameters of the offspring differ from those of the parent.

Importantly, either the sort or the parameter list of the parent is no longer limited to Accuracy and its associated parameters, but can be any meaningful ones, and the method can still be invoked. For example, Figure 4.12 shows how the method can be applied to a modifiability goal.

Figure 4.13 depict the three parameter decomposition methods with increasing degree of genericity: maximize-account-accuracy, accuracy-subclass and subclass methods.

Another example of a fairly generic parameter decomposition method is the attribute method. This method takes a parent having any sort type as its sort and a class as parameter, and generates a set of offspring, each having as its parameter one of the attributes of the parent parameter:

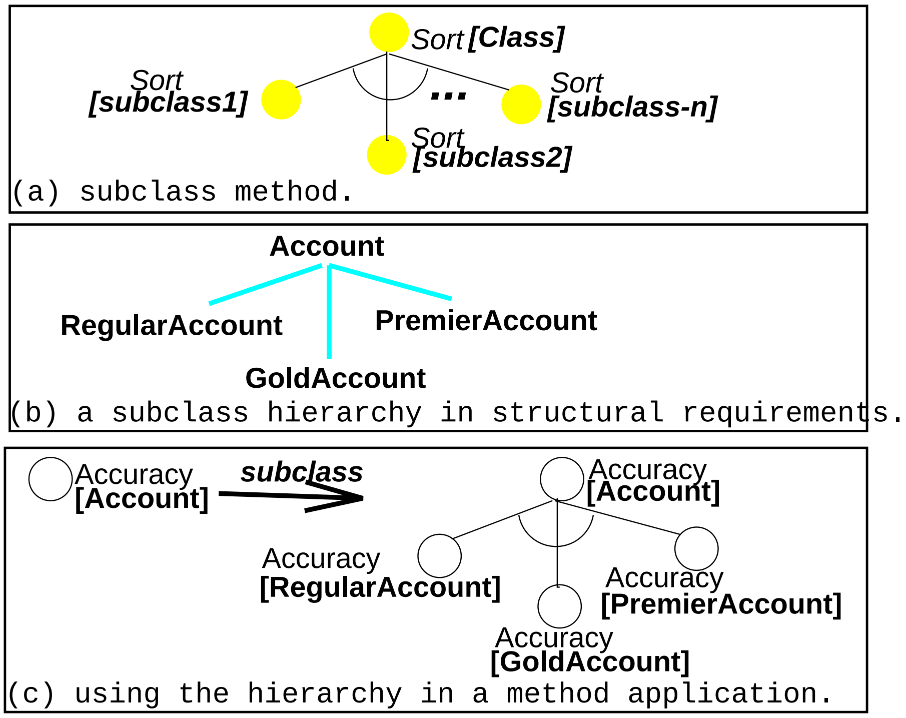

Figure 4.11 The subclass method and one of its applications.

DecompositionMethod attribute

parent: $s\ [p]$

offspring: $\{s\ [p_1],\ s[p_2],\ \cdots,\ s[p_n]\}$

contribution type: AND

applicabilityCondition: /* s must be a sort and p a class */

constraint: i p_i attributeOf p and

/* set up one offspring for every attribute
of the parent parameter class */

This method can be applied to any softgoal having any sort and any parameter list: Flexibility[UserInterface], Enhanceability[Component], OperatingCost[manpower], Accuracy[GoldAccount], etc. Taking one of these as an example, satisficing the goal of Accuracy[GoldAccount] requires that all attributes of gold accounts be maintained accurately. Figure 4.14 shows how to

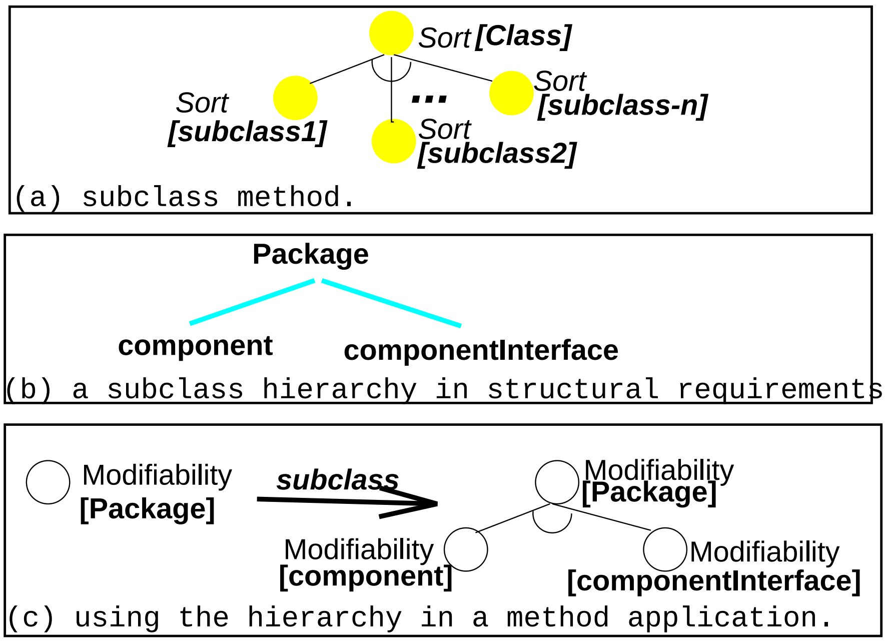

Figure 4.12 The subclass method and one of its applications.

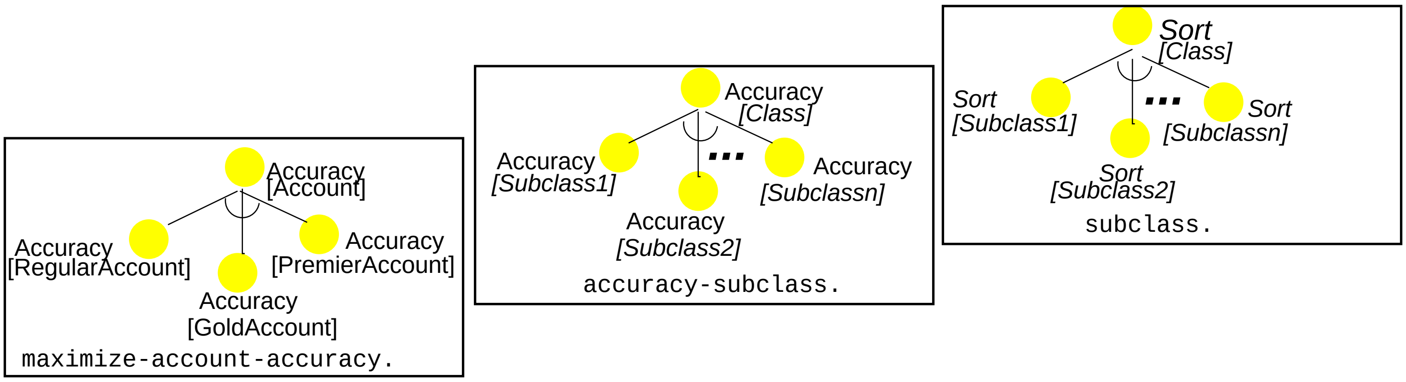

Figure 4.13 Three parameter decomposition methods with increasing degree of genericity.

be accomplish this by an application of the method. In the figure, gold card

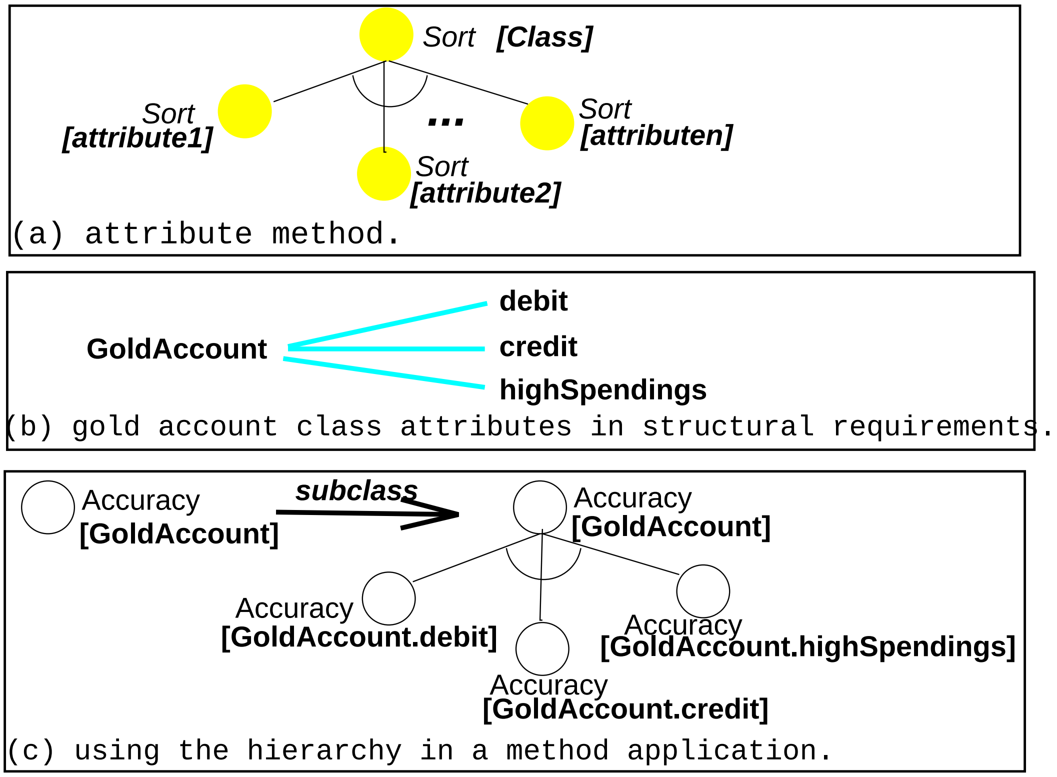

Figure 4.14 The attribute method and one of its applications.

accounts are assumed to have an attribute highSpending in addition to those of Account such as debit and credit.

Figure 4.15 illustrates how the developer might use the two decomposition methods, firstly the subclass method and next the attribute method, to refine the top-level goal Accuracy[Account].

4.3 OPERATIONALIZATION METHODS

Suppose the developer has satisfactorily clarified the initial non-functional requirements in terms of more detailed ones, either manually or using a catalogue of methods, or both.

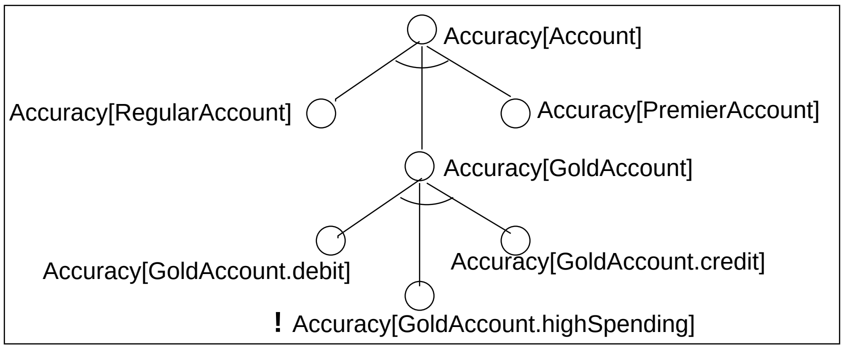

Figure 4.15 Softgoal contribution graph with two method applications.

S/he can now move on to trying to satisfice them. As with decomposition methods, the knowhow for satisficing non-functional requirements can also be captured and encoded as methods, and compiled into a catalogue. This way, the time and effort to search for relevant knowhow can be reduced, and a catalogue of such knowhow is made widely available and reusable.

Operationalization methods refine NFR goals into operationalized goals, basically design or implementation components, thereby committing the design that is being generated to particular design decisions.

As described in Section 3.2.2, the design can be a software architectural design, detailed design or implementation in the sense of the software lifecycle. But more broadly, the design can be just about anything that involves a decision-making process: a process specification and design, a project plan, a test plan, a customer communication plan, a change plan, a system specification, a requirements specification, a prototype, the result of risk analysis, the result of a CASE tool analysis, etc.

Operationalization methods can also refine operationalized goals into more specific ones, as operationalized goals can be coarse-grained and there can be more than one way to realize them in the target.

Thus, operationalization methods take either a NFR goal or an operationalized goal and generate one or more operationalized goals.

Returning to our example, consider the accuracy of information about various high spendings in gold card accounts, i.e., Accuracy [GoldAccount.highSpending].

The accuracy of this information might be ensured through periodic auditing of those databases, if the high spending record of each gold account is obtained from existing databases. This knowledge might be encoded as an operationalization method:

OperationalizationMethod auditing

parent: Accuracy $[x]$

offspring: Audit$[x]$

contribution type: +UNKNOWN

applicabilityCondition: InformationItem(x)

constraint: x in Database

Given an accuracy goal, this method will generate one offspring goal whose parameter is identical with the parent parameter. Figure 4.16 (a) and Figure 4.16 (b) respectively depict the definition of this method and one of its applications.

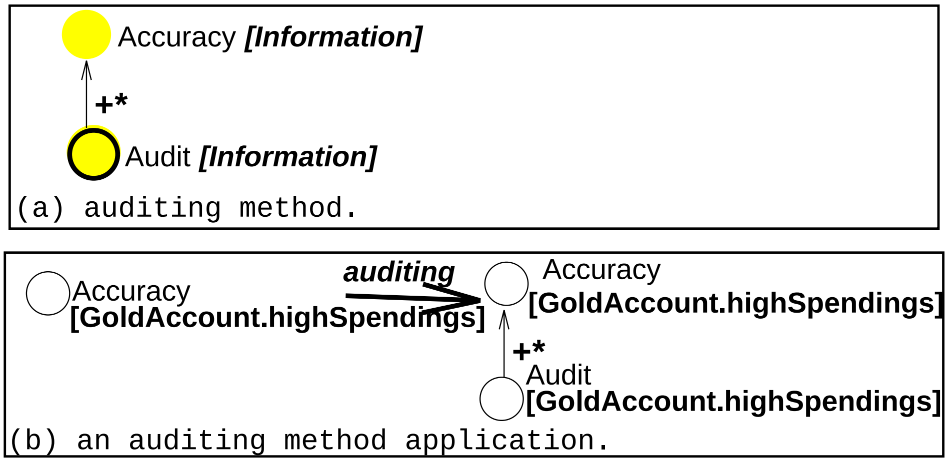

Figure 4.16 The auditing method and one of its applications.

If, on the other hand, information about various high spending reports are fed directly by an employee of the credit card organization in question, a method may call for the validation of the information by the employee's manager:

OperationalizationMethod validation

parent: Accuracy$[x]$

offspring: Validation$[x]$

contribution type: MAKE

applicabilityCondition InformationItem(x)

constraint: /* user validates x */

Figure 4.17 (a) and Figure 4.17 (b) respectively depict the definition of this method and one of its applications.

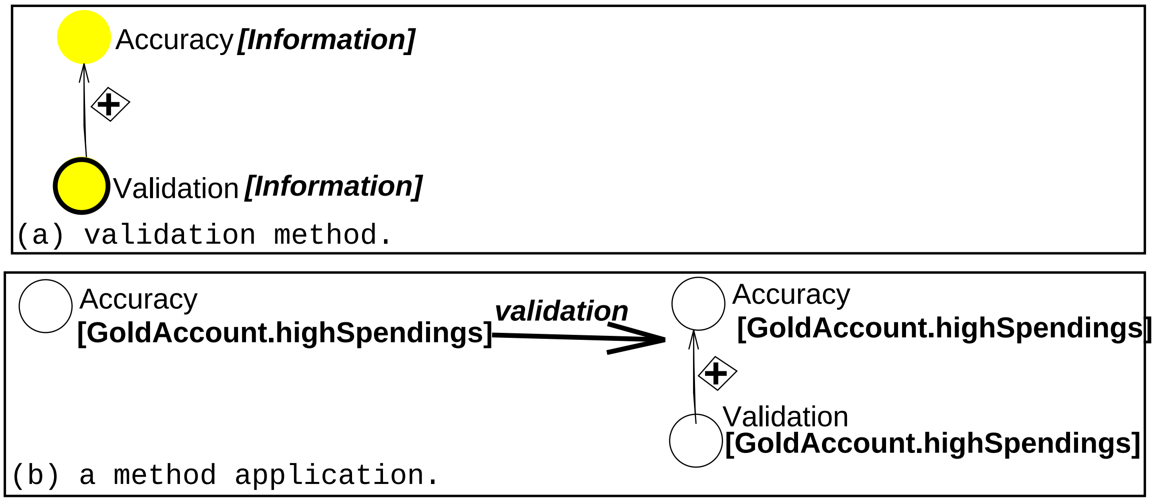

Figure 4.17 The validation method and one of its applications.

Using both of the above two methods, Accuracy [GoldAccount.highSpending] can be refined respectively to Audit [GoldAccount.highSpending] with +UNKNOWN contribution and Validation [GoldAccount.highSpending] with MAKE contribution (See Figure 4.20 for pictorial illustration).

Note that the designer may later change the type of the +UNKNOWN contribution once the design has proceeded further and it can be determined that auditing indeed leads to more accurate high spending 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 high spending information will have to be confirmed by another person, while auditing calls for the inclusion of an audit requirement on the database from which high spending data are imported.

The validation task can be assigned to appropriate people by instantiating the following method:

OperationalizationMethod validator-assignment
- parent: Validation[x]
- offspring: ValidatedBy[x', *Agent*]
- contribution type: HELP
- applicabilityCondition: x subsetOf x'
- constraint: *DesignerInput*(*Agent*)

where *designerInput*(*Agent*) refers to the particular agent which the designer specifies, such as class I secretaries. This is an example of the designer's involvement in generating offspring goals even after a method has been selected. During this interaction it is expected of the designer to consult domain experts to allocate an appropriate agent.

Figure 4.16 (a) and Figure 4.16 (b) respectively depict the definition of this method and how this method can be used to assign some people to the validation of gold accounts' high spending data.

As indicated in the beginning of this section, an operationalized goal can be refined, via a *decomposition* method, in terms of several other operationalized goals. The following decomposition method, validation-resource-availability is such a method, which can be applied to, for example, the above ValidatedBy[GoldAccount.highSpending, class-I-secretary]:

DecompositionMethod validation-resource-availability
- parent: ValidatedBy[*Information*, *Agent*]
- offspring: {Available[*Information*],
  Available[*policy-on-Information*],
  Available[*Agent*]}
- contribution type: AND

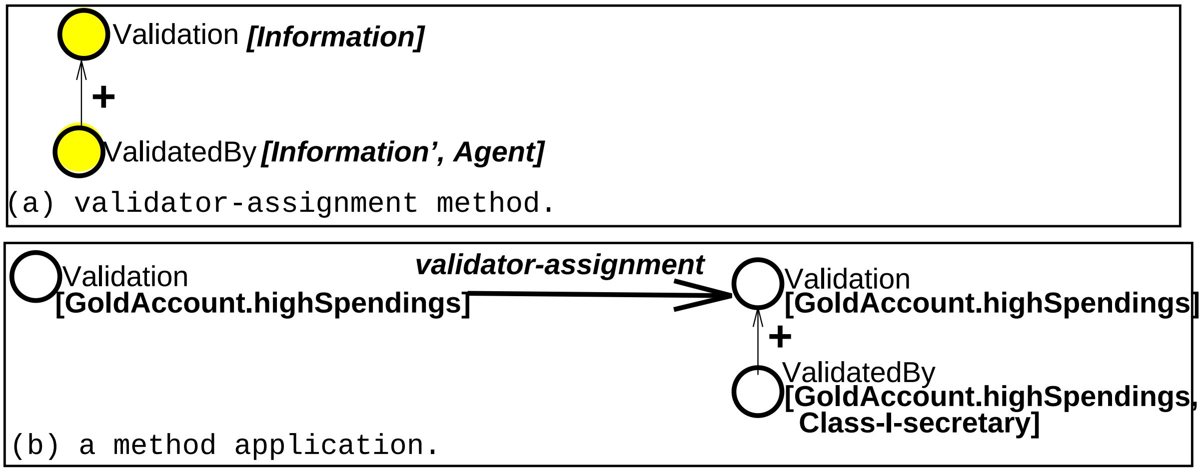

Figure 4.18 The validator-assignment method and one of its applications.

representing the requirement that, in order to perform the validation task, three resources should be available: information about the parameter, a policy which states predetermined standards on permissible values of the parameter, and an agent to perform the task.

Figure 4.19 pictorially depicts the validation-resource-availability method. Strictly speaking, this decomposition method is neither a parameter decomposition method nor a sort decomposition method. It is close to a parameter decomposition method as the parent parameters are inherited by two of the three offspring goals. Nonetheless, this method is more of a general decomposition method which generates offspring goals whose sorts and parameters have no direct relationship with those of the parent.

Figure 4.20 illustrates the softgoal contribution graph that might be generated by the simple example we have been introducing piecemeal. In the figure, four operationalization methods have been applied: one to Operating-Cost[manpower] to generate CasualUserInterface[Employee, GoldAccount.highSpending] and a +UNKNOWN contribution, two to !Accuracy[GoldAccount.highSpending] firstly to generate Auditing[GoldAccount.highSpending] with a +UNKNOWN contribution and secondly to generate Validation[GoldAccount.highSpending] with a MAKE contribution, and one to Validation[GoldAccount.highSpending] to generate ValidatedBy[GoldAccount.highSpending, class-I-secretary] with a HELP contribution.

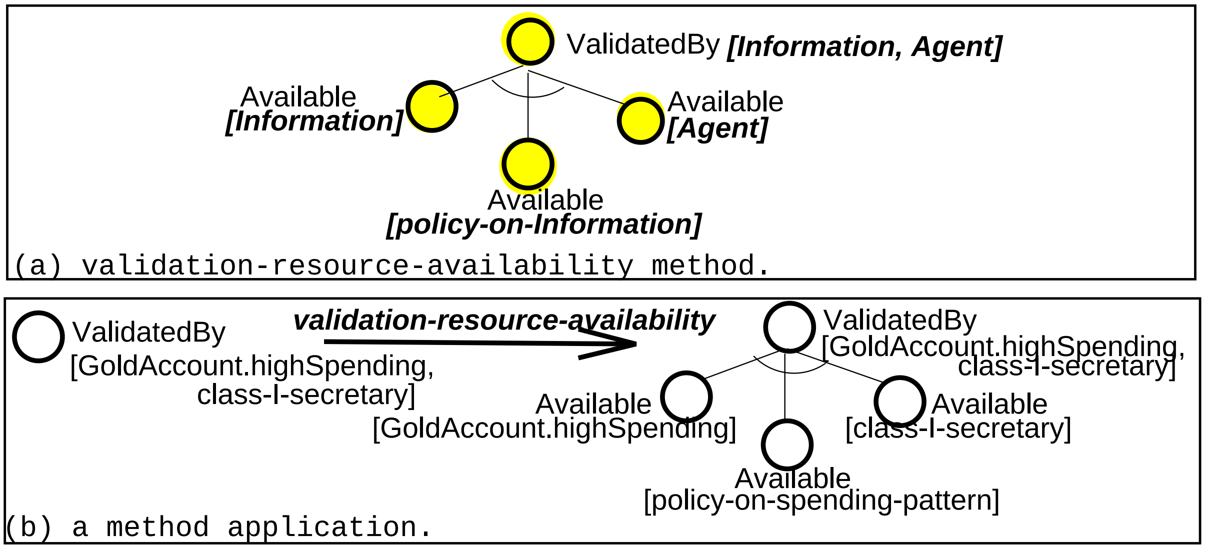

Figure 4.19 The validation-resource-availability method.

Although the current example shows that only one of the two design techniques is likely to be adopted for the satisficing of Accuracy[GoldAccount.highSpending], there can be a large number of operationalization methods that are applicable to a non-functional requirement all at the same time. If so, it is up to the developer to examine what impact such methods have on other types of non-functional requirements (e.g., cost, performance, production time and usability) and decide on what and how many operationalized goals to deploy.

4.4 ARGUMENTATION METHODS

Recall from Section 3.2.2 that claims are used to reflect domain characteristics into the decision-making process. This helps justify a variety of design decisions, namely, the way the initial NFR goals are elaborated, prioritized, and satisficed,

As with non-functional requirements and their operationalized goals, the knowhow for different ways of making design decisions and justifying them can be captured, encoded and compiled into a catalogue. Once catalogued, such knowhow too can be made widely available and reusable.

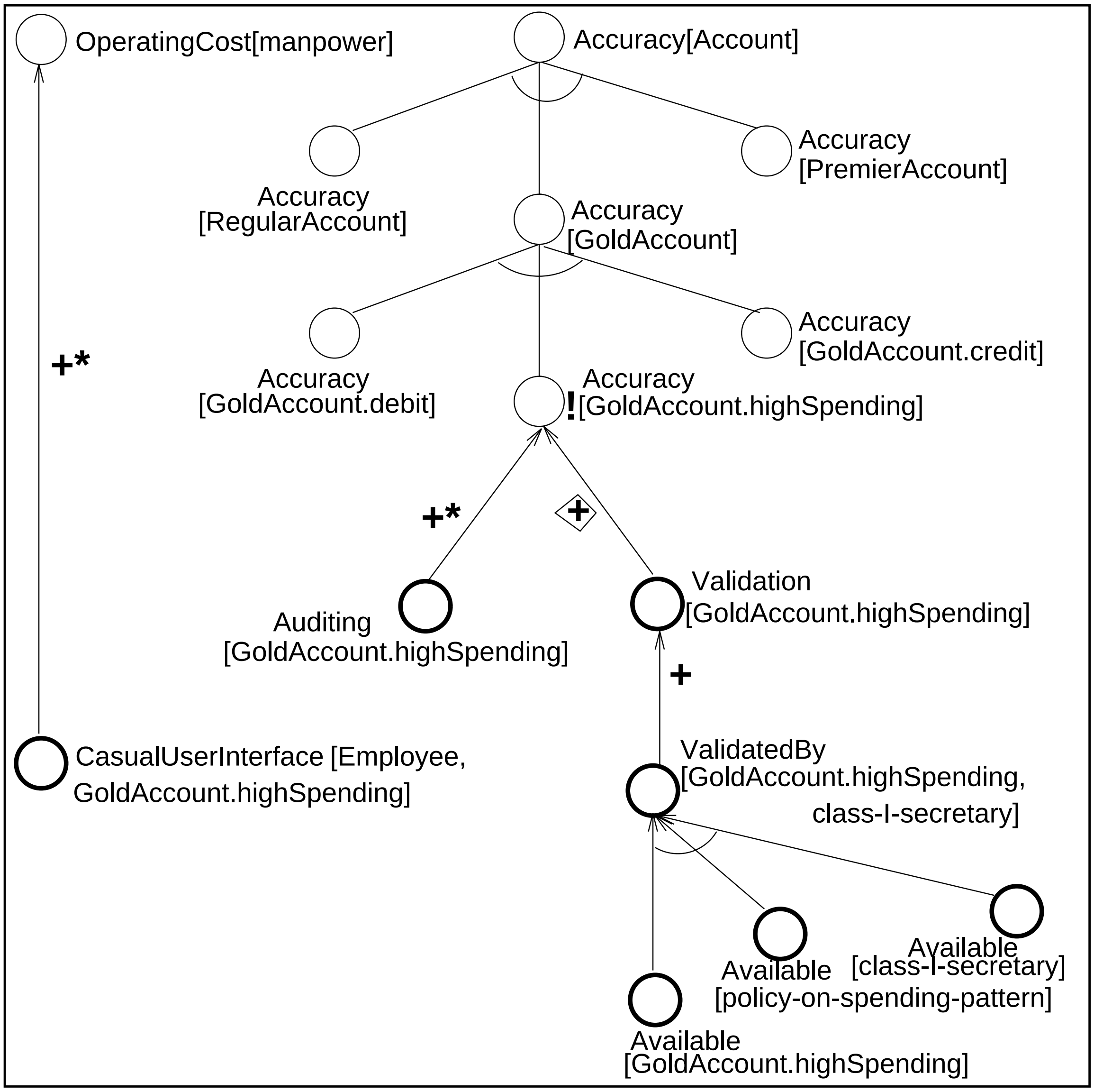

Figure 4.20 Softgoal Contribution Graph with applications of decomposition and operationalization methods.

Basically, argumentation methods refine a softgoal or a contribution into a claim, thereby indicating evidence, or counter-evidence, for the satisficing of the softgoal or contribution.

For instance, assigning class II secretaries to validating gold accounts' high spending data can be refined into, thereby being justified by, a claim requiring that the validation should be performed by an employee whose job is classified as higher than class II secretary.

ArgumentationMethod validator-assignment-justification
  parent: Argument [HELP (Validation[*Information*],
      ValidatedBy[*Information', Agent*])]
  offspring: *s* [*expr*]
  contribution type: MAKE
  applicabilityCondition: InformationItem (*Information*)
constraint: /* *expr* is about *Agent* performing validation of *Information* */

Note here that the parent is a contribution:

HELP (Validation[*Information*], ValidatedBy[*Information', Agent*])

In other words, the validator-assignment-justification method can be applied to a contribution of the above kind and generate a claim, where *s* and *expr* are replaced respectively by a particular sort and a formally-stated or informally-stated expression.

For example, this method can be applied to justify the assignment of class II secretaries to the validation of gold accounts' high spendings data. Figure 4.21 (a) and Figure 4.21 (b) respectively depict the definition of this method and one of its applications. As before, operationalized goals are denoted by thick circles, while claims by rectangles.

A formal version of the claim is given below:

Claim[exists (*employee, status*) where
  ValidatedBy[GoldAccount.highSpending, *employee*] and
      EmployeeStatus (*employee, status*) and
      HigherJobClassification (*status*, class-II-secretary)]

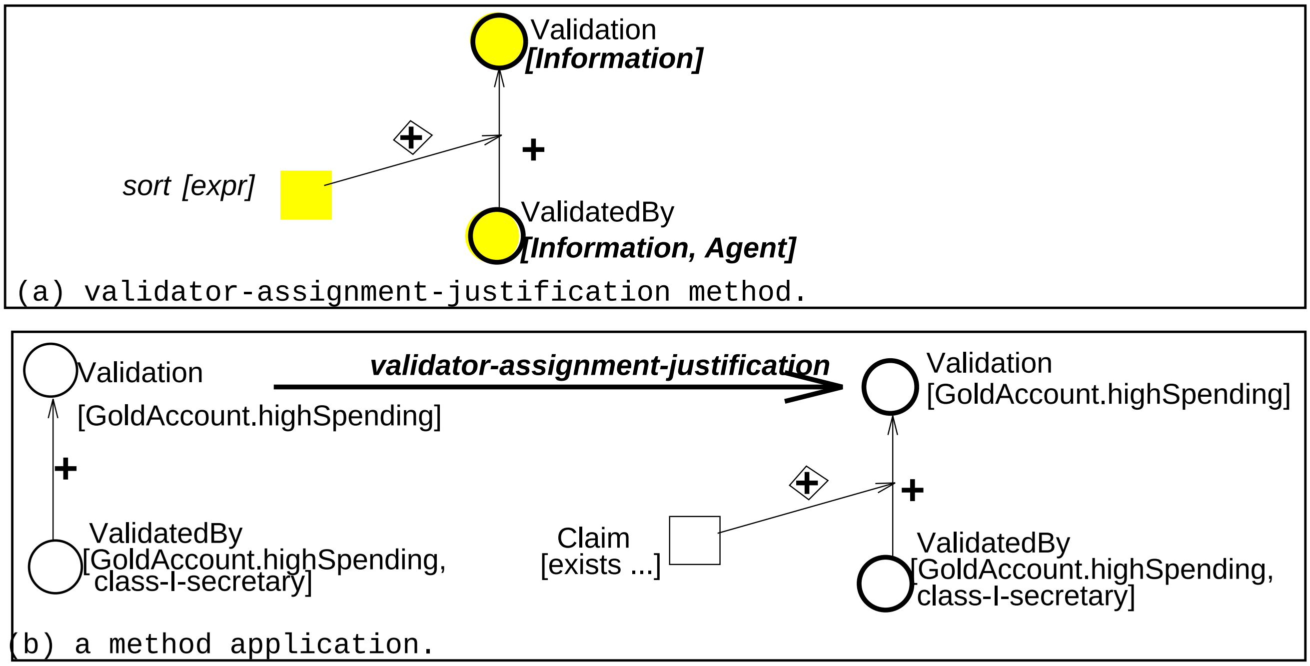

Figure 4.21 The validator-assignment-justification method and one of its applications.

This formal claim supports the refinement from the softgoal of validating high spending data to the one assigning a class I secretary (Sec I) to the task (instead of a class II secretary or below), by claiming that the validation should be performed by an employee whose job is classified as higher than class II secretary.

Unlike the above example which involves a contribution as the parent, the following shows how an argumentation method can be applied to an argument:

ArgumentationMethod validator-assignment-justification-support
parent: Claim[*expr*]
offspring: Claim [*expr'*]
contribution type: MAKE
applicabilityCondition: /* *expr* involves a validator assignment */
constraint: /* *expr'* involves support for the validator assignment */

This method can generate a claim supporting the previous formal claim by pointing out why high spending data should be validated by an employee whose job is classified as higher than class II secretary.

Figure 4.22 (a) and Figure 4.22 (b) respectively depict the definition of this method and one of its applications.

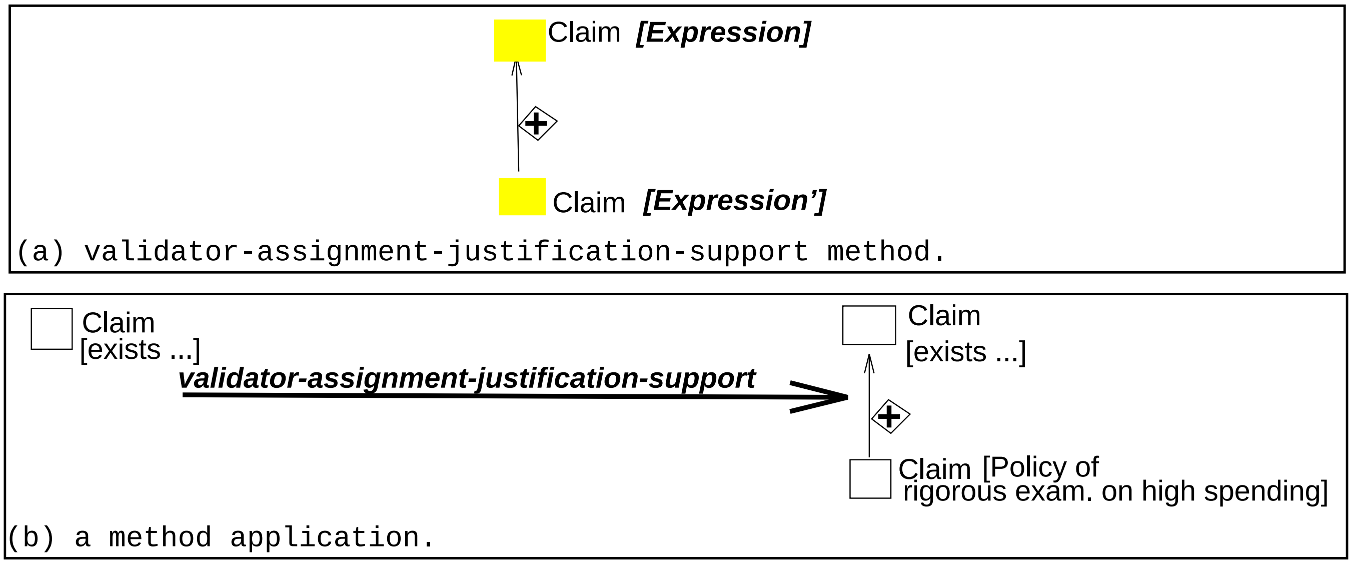

Figure 4.22 The validator-assignment-justification-support method and one of its applications.

An argumentation method can also be used to capture generic knowledge about selecting among alternatives. One such method was introduced earlier in Chapter 2, termed the *vitalFewTrivialMany* method. This argumentation method allows for putting more emphasis on important softgoals or contributions, during their elaboration and satisficing, during conflict resolution, and during selection among competing alternatives.

One variant of this method might be represented as:

ArgumentationMethod vitalFew
    parent: $s\ [p]$
    offspring: $s'\ [p']$

contribution type: +UNKNOWN
applicabilityCondition: /* parent is either a softgoal or contribution */
constraint: /* *s'* is Claim or Argument */ /* *p'* says the parent is one of the vital few */

Similarly, another variant of the *vitalFewTrivial* method might be represented, but this time with the BREAK contribution, as:

ArgumentationMethod trivialMany
parent: s [p]
offspring: s' [p']
contribution type: BREAK
applicabilityCondition: /* parent is either a softgoal or contribution */
constraint: /* *s'* is Claim or Argument */
/* *p'* says the parent is one of the trivial many */

Assuming that the operating cost goal is treated less important than the accuracy goal of high spending information, this method then can be used to explain, for example, why casual users (say employees in the credit card management system) should not be allowed access to such information:

Figure 4.23 (a) and Figure 4.23 (b) respectively depict the definition of this method and one of its applications. As operating cost is less important than accuracy of high spending data, casual-user interface would be disallowed.

Figure 4.24 illustrates part of the softgoal contribution graph that might be generated by the argumentation methods above (assuming some operationalization methods have previously been applied). As before, operationalized goals are denoted by thick circles, while claims by rectangles. In the figure, four argumentation methods have been applied: two to contributions, and the other two to softgoals.

These argumentation methods help the developer to construct an *argumentative structure* as part of a softgoal contribution graph, i.e., nesting of arguments together with softgoals and contributions. In an argumentative structure, an argument can, on the one hand, offer positive or negative support for a particular softgoal or contribution, and in turn be supported or denied by other softgoals or contributions, on the other. This kind of power to organize

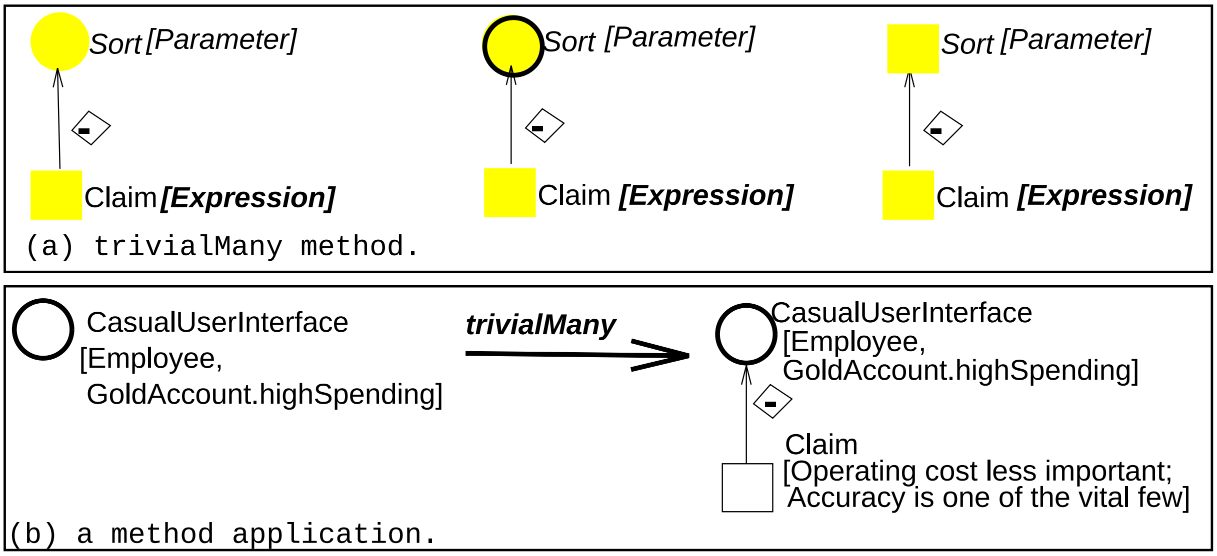

Figure 4.23 The trivialMany method and one of its applications.

design rationale stems from the ability of contributions which, in addition to relating between softgoals, can also relate between softgoals and contributions, as well as between contributions.

Figure 4.25 illustrates the softgoal contribution graph that might be generated by the simple example we have been introducing piecemeal. The contributions among these objects are shown through contributions on the left- and right-hand sides of Figure 4.25. Some contribution types are omitted from the figure.

4.5 CORRELATION RULES

The non-functional requirements set down for a particular system may be contradictory or harmonious. For instance, "faster" (production time) may be in conflict with "cheaper" (cost), if it involves hiring more software engineers. On the other hand, "better" (software) may be harmonious with "faster", if "better" (software) significantly cuts down testing and defect-correction time.

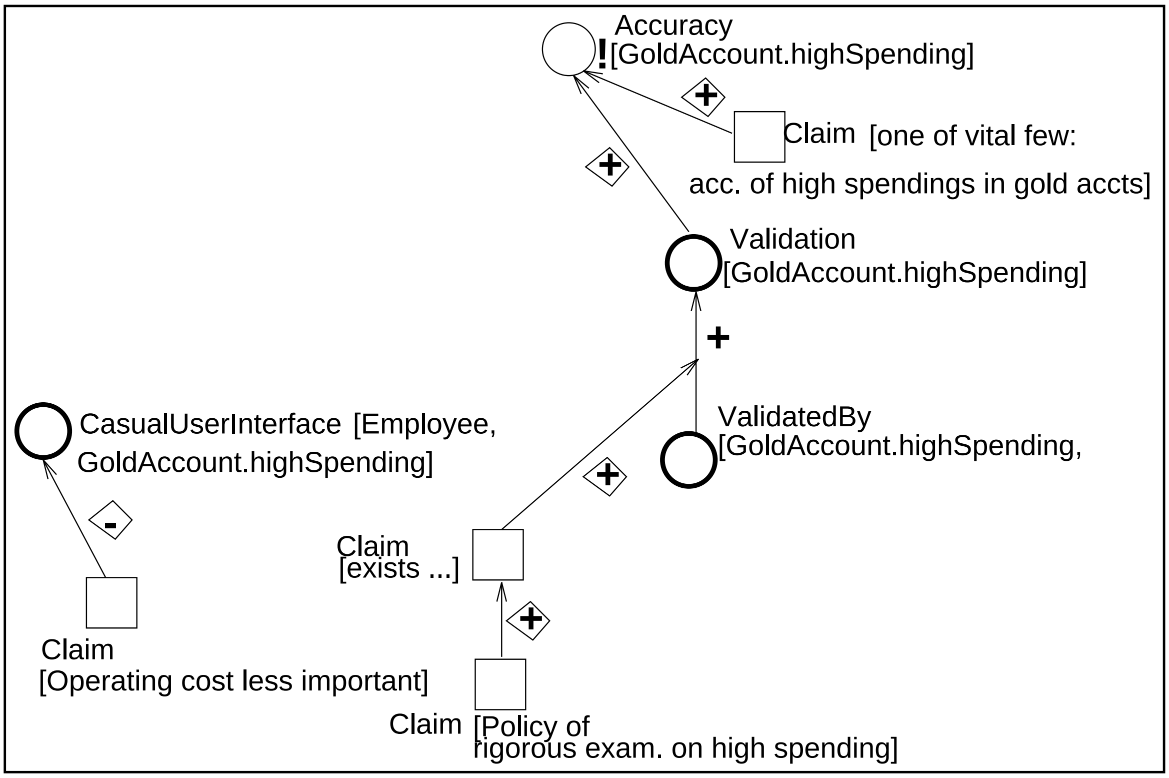

Figure 4.24 Part of a softgoal Contribution Graph with applications of operationalization and argumentation methods.

In dealing with non-functional requirements in software engineering, thus, it is of paramount importance for the developer to recognize and accept this kind of conflict and harmony as a natural phenomenon.

In the NFR framework, conflict and harmony are represented respectively by positive correlations and negative correlations. During the process of software development, then, interactions between non-functional requirements are discovered and indicated as such in a softgoal contribution graph.

Such a discovery can be made either manually by the developer on-the-fly or with the assistance of *correlation rules*. Just like methods, correlation rules allow for capturing knowledge about generic interactions between softgoals, for encoding such knowledge, and for compiling it into a catalogue. Initially

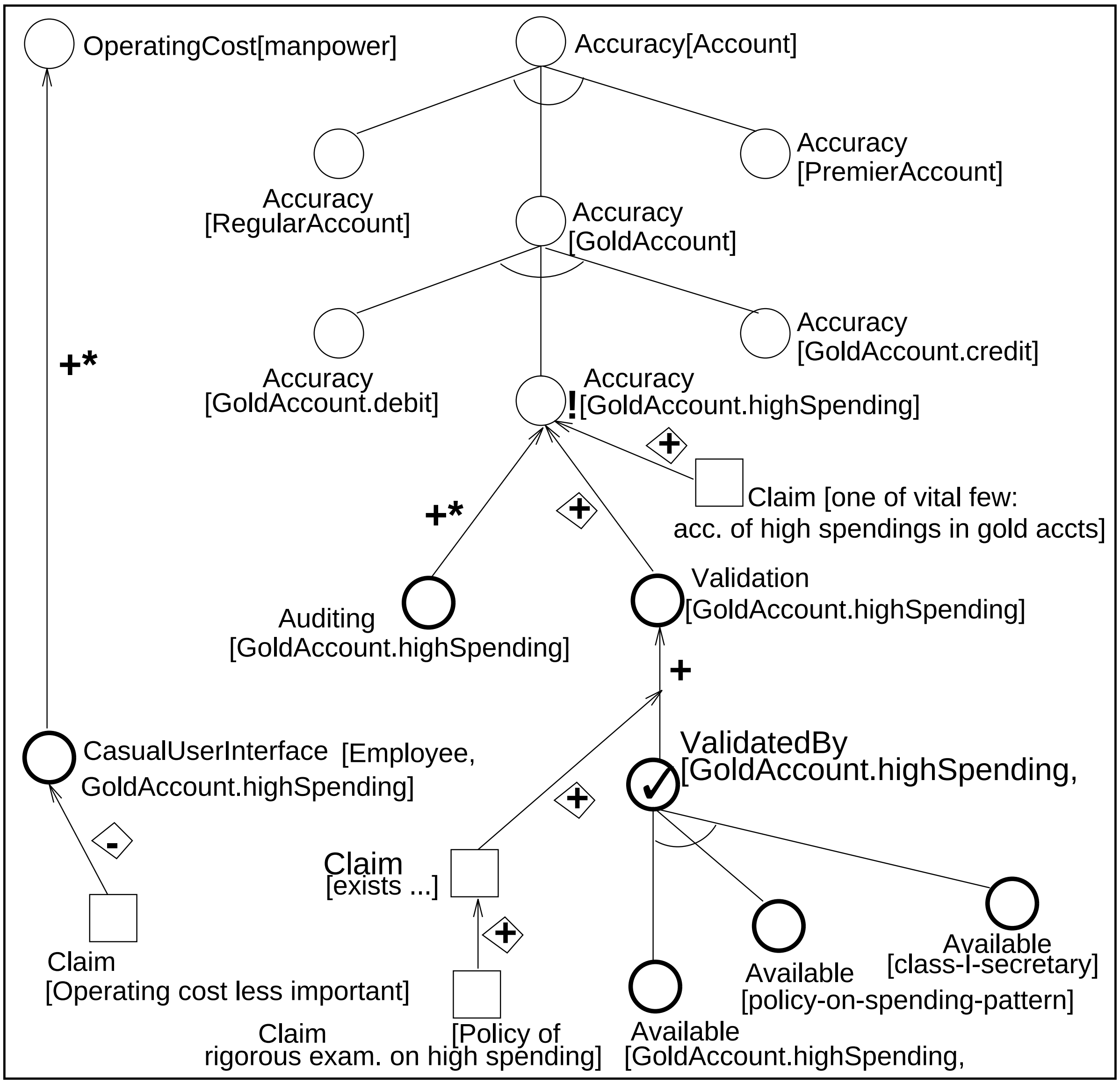

Figure 4.25 Softgoal contribution graph for accurate account attributes.

collected from the literature and industry experience, correlation rules then can easily be shared, extended, tailored, and reused.

Unlike methods, however, correlation rules are not usually selected and applied explicitly by the developer during the development process. Instead, they are *triggered* by certain events, such as generation of a design technique as the result of a method application or change in the criticality of a softgoal. Triggering can also be done explicitly by the designer.

When triggered, correlation rules are used by the tool detect correlations, both positive and negative. Of course, it is up to the designer to decide whether to incorporate them into the softgoal contribution graph.

Detecting negative correlations might be viewed as revealing the (hidden) duality of a design technique — it can be good for one goal but at the same time bad for another.

In this sense, a contribution generated as the result of a method application is termed a *direct contribution*, while a contribution inferred from a correlation rule is termed a *indirect contribution*.

When selecting among competing operationalized goals correlation rules about interactions between such goals and NFR goals can serve as a basis for performing trade-off analysis. Naturally, the developer might want to select those operationalized goals that yield the most benefits and the least sacrifices, if possible.

Returning to our credit card account management system, consider an operationalized goal whereby the system under design will offer an interface for "casual" users, say, all employees, who wish to query or update the system's database:

OperationalizationMethod casual-user-interface
  parent: OperatingCost[manpower, $100 000]
  offspring: CasualUserInterface[Employee, Database]
  contribution type: HELP

A similar technique was also mentioned earlier in Section 3.3.

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:

CorrelationRule casual-access-hurts-accuracy
parent: Accuracy [Database]
offspring: CasualUserInterface [Employee, Database]
contribution type: HURT

Given the two goals Accuracy[GoldAccount.highSpending] and CasualUserInterface[Employee, GoldAccount.highSpending], this rule then can be used to generate a *HURT* contribution between them. Figure 4.26 (a) and (b) respectively depict the definition of this correlation rule and its uses.

Interestingly, the same rule can be used to generate the same contribution, but this time when a casual interface goal is generated. Figure 4.26 (c) pictorially depicts this situation.

Similarly, the rule can also be used to generate the same contribution, but this time given only the accuracy goal. Figure 4.26 (d) pictorially depicts this situation.

These kinds of using a correlation rule are helpful in not only detecting negative correlations between those objects that are already in the softgoal contribution graph but proving a warning to the developer against omission of those NFRs that she might have omitted. These inferences can be confirmed by a domain expert, analyst, or the designer.

A more general rule can be stated for imposing constraints on the situations when a correlation can be triggered. For instance, the following is a more general correlation rule which allows for relating between softgoals with different parameters and for stating the upper bound on the number of casual users (say, five):

CorrelationRule many-casual-access-hurts-accuracy
parent: Accuracy [*Information*]
offspring: CasualUserInterface[*Agent, Information*]
contribution type HURT

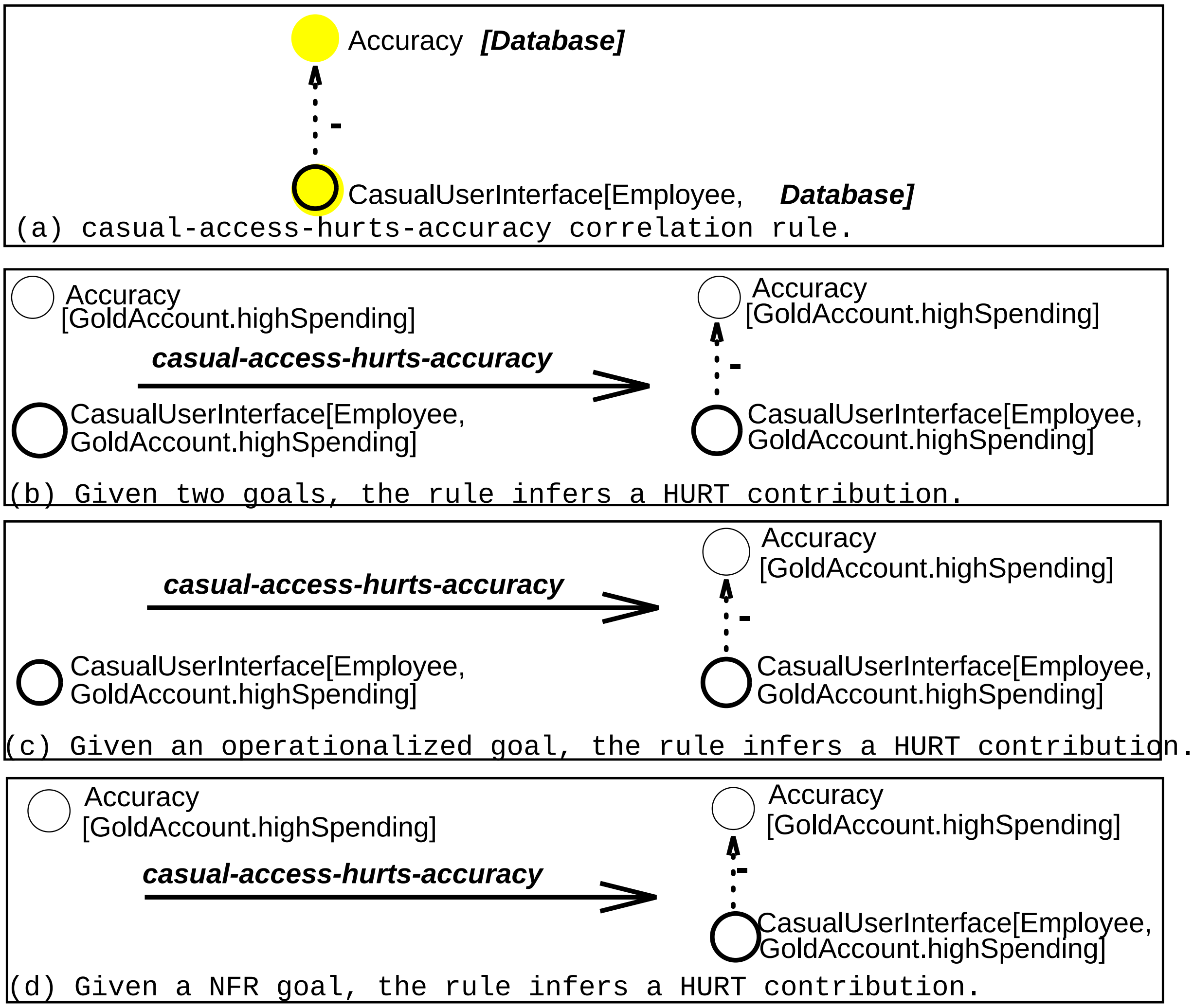

Figure 4.26 The definition, and three uses, of the correlation rule casual-access-hurts-accuracy.

triggerCondition:

cardinality $(Agent) > 5$ and

Information subsetOf *Information'*

/* casual access to information by more than five employees hurts accuracy */

As with the simple rule earlier, this rule can be used to generate a *HURT* contribution, here between the softgoals CasualUserInterface[Employee, Database] and Accuracy[GoldAccount.highSpending], when the triggerCondition is met, i.e., GoldAccount.highSpending subsetOf Database and there are more than five employees using the system.

Taking another example, consider a validation technique which is positive for Accuracy [GoldAccount.highSpending], where class II secretaries will be the validator. This operationalized goal would contribute negatively to a security goal which discourages secretaries from accessing gold accounts' high spending data, and vice versa:

CorrelationRule validation-of-the-info-by-the-owner-hurts-security
parent: Security [*Information*]
offspring: Validation [*Information*]
contribution type: BREAK
triggerCondition
ValidatedBy (*Information, employee*) and
not accessAllowed [*Information, employee*] and
/* validation violates security policy when done by
unauthorized employee */

While the above assumes validation of the same information by the same employee, the following is for a more general case relaxing the assumption:

CorrelationRule validation-hurts-security
parent: Security [*Information*]
offspring: Validation [*Information'*]
contributtion type: BREAK
triggerCondition
ValidatedBy (*Information, employee'*) and
not accessAllowed [*Information, employee*] and
Information' subsetOf *Information* and
hasLowerAccessRight(employee', employee)

/* validation violates security policy when done by
unauthorized employee. */

where accessAllowed[*Information, employee*] holds if employee is allowed access to *Information*, and hasLowerAccessRight[*employee', employee*] holds if the security scheme classifies *employee'* as having lower access right than *employee*.

A correlation exists not only between a NFR goal and an operationalized goal but two operationalized goals as well. For example, consider the case where two operationalized 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 techniques calling for auditing and validation of information may influence each other positively, through *HELP* or MAKE contributions. This might happen if there is no one on staff assigned to either task, which may not individually justify hiring additional staff member but jointly suggest the hiring of one new staff member. This situation is represented below by two correlation rules:

CorrelationRule validation-helps-auditing
parent: Audit[*Information*]
offspring: Validation[*Information'*]
contribution type: HELP
triggerCondition:
exists (*employee*) where
(ValidatedBy [*Information, employee*] and
(AuditedBy [*Information', employee*]
/* validation helps auditing synergistically */

CorrelationRule auditing-helps-validation
parent: Validation[*Information*]
offspring: Audit[*Information'*]
contribution type: HELP
triggerCondition:
exists (*employee*) where
(ValidatedBy (*Information, employee*) and

(AuditedBy (*Information', employee*)
/* auditing helps validation synergistically */

This situation is depicted in Figure 4.27. In the figure, the triangu-

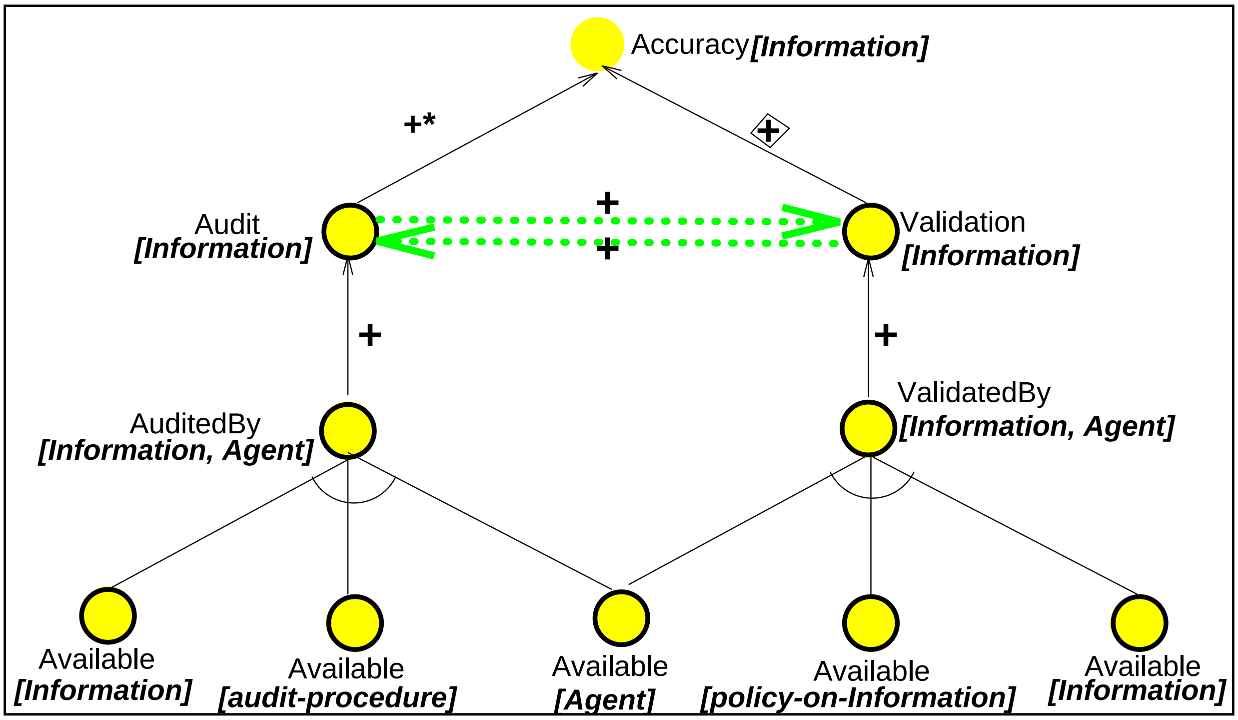

Figure 4.27 The synergy between validation and auditing.

lar shape represents those parts of parent, offspring and contribution type in the above two correlation rules. The lower part depicts the triggerCondition, assuming that AuditedBy[*Information, Agent*] and ValidatedBy[*Information, Agent*] are decomposed via resource-availability methods such as the validation-resource-availability method described in Section 4.3.

Note that both validation and auditing have a common offspring, Available[Agent]. Another example of why the structure of softgoals and contributions is not a tree but a graph.

On the other hand, it is also possible that the two softgoals influence each other negatively, through *BREAK* contributions, if only one new staff member can be hired and yet each of the two tasks must be handled by a different staff member.

Finally, consider having built-in procedures for validating, or auditing, the data managed by the information system in order to enhance accuracy of the data. However, such procedures in general require additional manpower, thereby interfering with operating cost requirements. For instance, a built-in procedure for validation might interfere with OperatingCost[manpower, $100 000]. This interference might be represented by a correlation rule:

CorrelationRule validation-hurts-operatingCost
  parent: OperatingCost [manpower, $100 000]
  offspring: Validation [*Information*]
  contribution type: HURT

As the number of correlation rules grows, it becomes harder to examine cross-impacts among softgoals, during tradeoff analysis and subsequently during selection among competing techniques. As an aid to the developer, for this reason, a synopsis of correlation rules might be created. Table 4.1 shows a synopsis of some of the correlation rules described in this section.

NFR **OperationalizedGoal**	*Accuracy*	*Security*	*OperatingCost*
Validation	<C_1, HELP>	<C_4, BREAK>	< TRUE, BREAK>
Auditing	<C_3, +UNKNOWN>	< TRUE, HELP>	< TRUE, HURT>
CasualUserInterface	< TRUE, HURT>		< TRUE, HELP>

Table 4.1 A correlation table of a synopsis of correlation rules.

In Table 4.1, each C_i stands for the triggerCondition associated with the rule in each entry; if TRUE, it always holds.

Figure 4.28 illustrates the use of some of the correlation rules in this section.

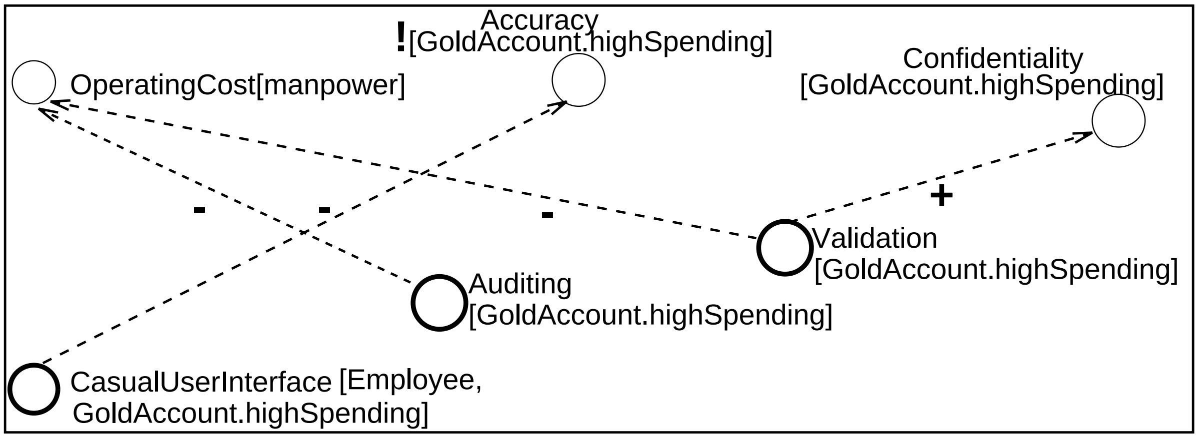

Figure 4.28 Softgoal Contribution Graph with correlations.

4.6 PUTTING THEM ALL TOGETHER: THE GOAL-DRIVEN PROCESS

In the previous and current chapters, the five components of the NFR framework have been presented. For each component, some illustrations have been provided in an incremental fashion with the use of a softgoal contribution graph.

It is now possible to describe more fully the process of dealing with non-functional requirements, in the context of system development. The description generalizes the process, given earlier in Chapter 2 which illustrated only a sample process using the credit card management system as the example.

The process starts with an initial set of high-level non-functional requirements as goals, along with an initial set of functional requirements. It iteratively refines the softgoals into more specific ones, while establishing contributions, namely, relationships between softgoals (and among softgoals and contributions).

Figure 4.29 illustrates the softgoal contribution graph that might be generated by the simple example we have been introducing piecemeal. This figure is identical with Figure 3.? in Chapter 3, except for the contribution between the

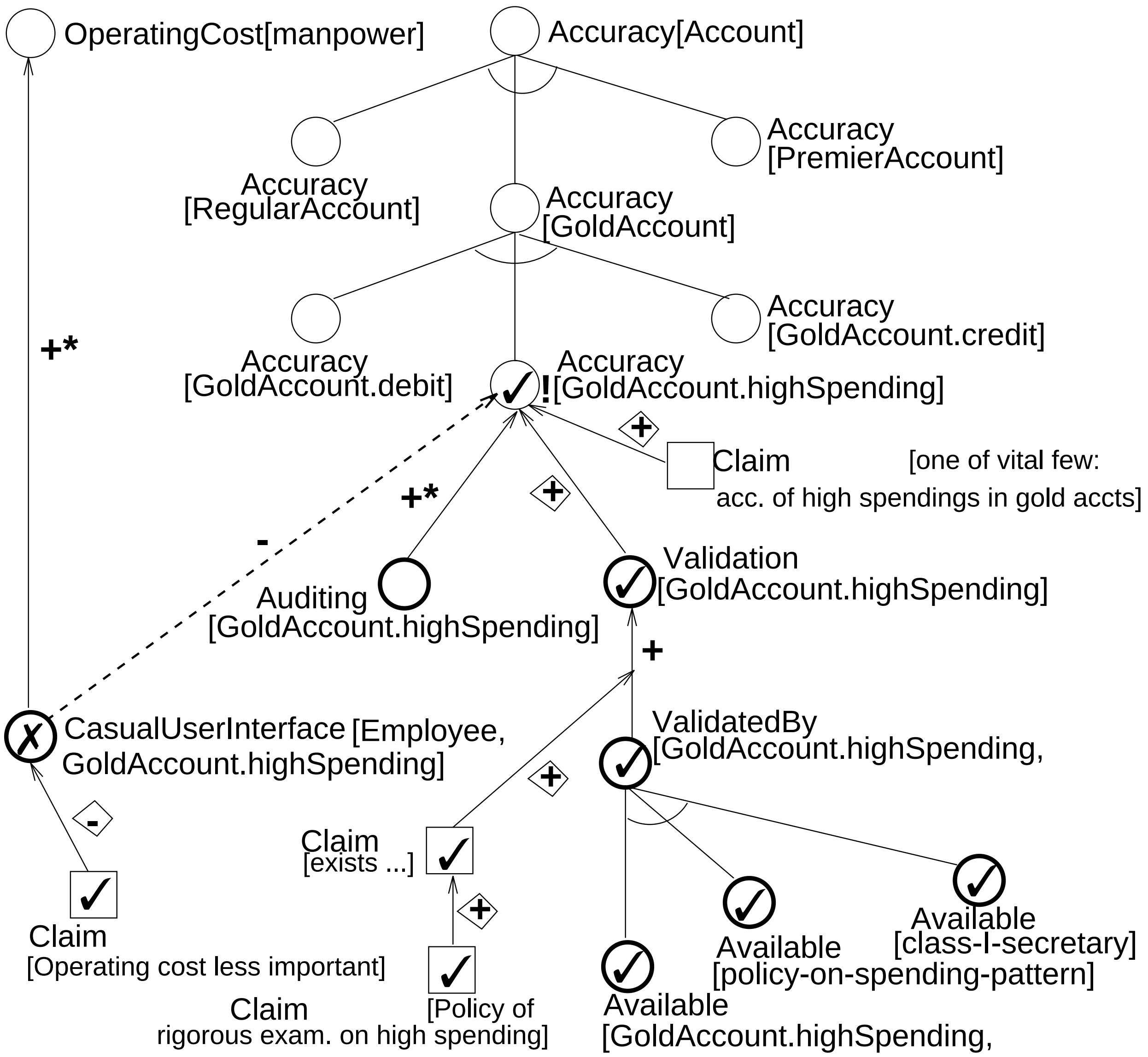

Figure 4.29 Softgoal contribution graph for accurate account attributes.

CasualUserInterface operationalized goal and !Accuracy[GoldAccount.highSpending]. Here, the contribution is indicated as an indirect one.

As described in Sections 3.5 and 4.1, functional requirements, both structural and behavioral, serve as parameters of softgoals and contributions. As a result, functional requirements too may need to be elicited, analyzed and specified, during the process of iterative refinements. For example, an application of the subclass method may necessitate the introduction of new (sub)classes if they are not already in the structural requirements description.

Throughout this process, softgoals and contributions are organized in a softgoal contribution graph, and partitioned into *Open* and *Closed*, which respectively represent a list of all softgoals and contributions that are to be refined and that have been completely refined.

In the process, the developer is in full control, selectively focusing attention and determining what softgoal or contribution to refine next. The developer is also in full control, determining how to refine the chosen object and how much to refine.

After selecting a softgoal or contribution from *Open* for refinement, the developer can apply one of the pre-compiled methods, if available; otherwise, she can propose a refinement of her own. Carrying out the chosen refinement on the selected softgoal or contribution, the parent, results in the generation of its offspring, as well as a contribution between the parent and the offspring. The newly-created offspring and contribution are then added to *Open*.

In particular, use of pre-compiled methods implies: firstly, browsing a catalogue of methods which are applicable to the parent; secondly, selecting a particular method; thirdly, ensuring firstly that the sort and parameter list of the chosen softgoal or contribution match (or are related to) those in the parent part of the definition of the selected method, and secondly that the current situation meets the selection criteria given in the definition of the method; and lastly generating the offspring and contribution, according to the offspring and condition parts of the definition of the chosen method.

In consideration of design tradeoffs, correlations are then introduced between the new softgoals, or contribution, and the already-existing ones. This involves using both the developer's judgement and correlation rules, and results in the creation of new contributions between the new one and old ones, and their addition to *Open*.

Use of pre-compiled correlation rules implies: firstly, using the trigger conditions associated with the rules to test if any correlation rules are applicable to the newly-added softgoal or contributions of *Open*; secondly, inferring contributions, both positive and negative, between the existing softgoals or contributions and the newly-created ones; and lastly establishing such contributions in a softgoal contribution graph. Once established, the contributions are confirmed or rejected by the developer.

At any point during this process, the developer may prioritize the softgoals and contributions, in either *Open* or *Closed*, by assigning values to their criticality or priority attribute. Prioritization may be needed due to resource limitation, bounded rationality, domain constrains, etc. For example, due to limited development time available, the developer may need to spend more effort and time on critical goals than on less-critical ones (e.g., more for an accuracy goal on high spendings than an accuracy goal on insignificant amounts).

Prioritization can also help resolve conflicts In Figure 4.29, for example, Validation is good for accuracy but hurts operating cost. Here, the technique is adopted as the accuracy goal is more important then the operating cost goal. Additionally, selection among alternative techniques may also be based on their relative importance, and analysis of arguments on their relative weights (e.g., distinguishing major policies from minor ones), etc.

This process of refinement, selection among alternatives, analysis of tradeoffs and prioritization is repeated for the chosen softgoal or contribution until there are no more refinements the system or the designer can offer. At this time, the softgoal or contribution is placed on the *Closed* list and another open softgoal or contribution is selected.

A second alternative for softgoal or contribution refinement is to simply label the softgoal or contribution *satisficed* or *denied*. Such evaluation may come about either because of input from the designer or because of the use of a method or correlation rule during refinement. Upon the change in any label, the interactive evaluation procedure determines, with the input from the developer, all those softgoals and contributions that are affected by such a change, and assigns new labels to them appropriately. A closed softgoal or contribution should have as its label S or D, while an open one can have any of the other label values.

Thus, the NFR framework takes a a goal-oriented approach to dealing with non-functional requirements. This approach makes it possible to systematically treat non-functional requirements and ultimately use them as the criteria for making selections among competing target alternatives. in complementary to an evaluation-centric, product-oriented approach, in software engineering.

4.7 DISCUSSION

4.7.1 Key Points

Of the five components which the NFR framework offers, this chapter has presented *methods* and *correlation rules* which Chapter 3 did not cover.

These two components help construct a softgoal contribution graph in which softgoals and contributions, together with their labels, are organized as a record of design intentions, alternatives, tradeoffs, decisions and rationale — the focus of Chapter 3.

Methods are a means to collect, encode and catalogue knowledge about clarifying the meaning of softgoals, satisficing them and providing arguments in support or denial of development decisions. Correspondingly, the NFR framework offers three types of methods: decomposition methods, operationalization methods and argumentation methods. Once catalogued, methods can be tailored, improved, extended and reused.

Similarly, correlation rules are a means to collect, encode and catalogue knowledge about interactions, synergistic or harmonious, between and among softgoals and contributions. Correlation rules are useful particularly during the analysis of tradeoffs among alternative techniques and discovery of omission of softgoals (non-functional requirements or operationalized goals). Just like methods, methods can be tailored, improved, extended and reused.

4.7.2 Sources of Literature

The representation of security requirements is adopted from the area of computer security (e.g., [Hartson76]).

The *vitalFewTrivialMany* method has its origin from Juran [Juran64] who coined the terms *vital few* and *trivial many* as applied to the Pareto principle, after the Italian economist, Vilfredo Pareto, who quantified the extent of inequality or nonuniformity of the distribution of wealth. Juran later noted in

[Juran79] (the first edition came out in 1951) that it was a mistake to name it the Pareto principle, and that the principle should have been identified with M. O. Lorenz, who had used curves to depict concentration of wealth in graphic forms. McCabe et al. [McCabe87] applied this principle to software quality activities. This also reflects the 80–20 rule that 80% of development effort or software improvement results from 20% of the software under development or in the system [Boehm87].

The correlation table is similar, in spirit, to the "relationship matrix" in [Hauser88], which indicates, in terms of sorts and four types of values (strong positive, medium positive, medium negative, and strong negative), how much each engineering characteristic affects each customer quality requirement. Unlike the NFR framework, the House of Quality by Hauser et al. deals with only actual design instances, rather than offering generic methods with parameters or generic correlation rules with conditions which facilitate reuse of design knowledge. Another difference is that, via a softgoal contribution graph, the NFR framework allows for justifications of design decisions to be tightly associated with decision points.

4.8 RELATED WORK FOR THE FRAMEWORK

The characteristics of the framework to be proposed in this book include:

- The focus of the work is *large systems* development rather than programming-in-the-small;
- Requirements specifications consist of *both functional and non-functional requirements*;
- The solution takes a process-oriented approach, where non-functional requirements are treated as goals;
- This work adopts the ideas in recent work on decision support systems and the DAIDA environment.

Little work can be found that aims specifically to provide systematic methodologies for generating large systems from requirements that consist of both functional and non-functional requirements. However, there are several lines of work which address certain aspects of the framework to be proposed in this book. Related work is discussed in three parts: i) Since quality control issues are relevant to non-functional requirements and have been dealt with in those areas of management, manufacturing, software, and information system development, we discuss projects and approaches in such areas; and ii) Since we are concerned with developing systems, we discuss existing tools, methodologies, frameworks, and environments used for developing software systems. iii) Since we treat non-functional requirements as goals, we discuss goal-oriented approaches in several different areas;

Quality Assurance and Control

The field of quality control is rich in general guidelines and statistical measurement techniques that are used on a daily basis. Although related to each other and sometimes used interchangeably, quality assurance is regarded as being broader in scope than quality control. According to ISO 8402, quality assurance refers to the overall management function that determines and implements the quality policy, which is established by senior management as the overall quality intentions and objectives of an organization. According to ISO 9001, quality control is concerned mostly with the latter part of quality assurance, ensuring firstly the use of appropriate tools in producing the final product and secondly that the final product meets pre-determined standards.

One of the most advanced methodologies for quality control, described by Hauser et al. [Hauser88], aims to reflect customer attributes in different phases of automobile design, by helping establish and visualize relations between manufacturing functions and customer satisfaction. In this kind of so-called Quality Function Deployment methodology, customer attributes are (recursively) treated as bundles of more finer-grained attributes and customers' evaluations of their relative importance are registered. The relationship between these customer attributes and engineering characteristics is then expressed in terms of different degrees of scale, along with the relationship between different engineering characteristics. Finally information associated with engineering characteristics, such as technical difficulties, relative weights, and cost, are recorded and used in making design decisions.

For strategic quality analysis and management, Garvin [Garvin87] proposes eight critical dimensions or categories of quality: performance, features, reliability, conformance, durability, serviceability, aesthetics, and perceived quality. He states that some dimensions have more objective standards than some other ones, stating that performance has more objective standard than perceived quality. He also states that some dimensions are always mutually reinforcing, while some other ones are not. As an improvement in one may be achieved only at the expense of another, a product or service can rank high on one dimension of quality and low on another. This interplay is noted as the key that makes strategic quality management possible and suggests competing on selected dimensions.

Among the eight dimensions, robustness, which Garvin treats as a component of conformity, has drawn a significant interest in the area of manufacturing design. Taguchi et al. [Taguchi90] propose a statistical approach to dealing with robustness, where one major emphasis lies in taking into account of potentially critical interactions among design components. This proposal emphasizes that quality loss starts from the time a product is shipped and should include warranty costs and nonrepeating customers. This proposal also formulates that quality loss increases by the square of deviation of design components from the target value.

A more focused area than the above and of interest to us is *software quality assurance* (*SQA*), which is defined in ANSI/IEEE Standard 729–1983:

> "In software system engineering, a planned and systematic pattern of all actions necessary to provide adequate confidence that the software and the delivered documentation conforms to established technical requirements."

Previous work in this area has focused primarily on managerial issues, quantitative measurements or product testing and inspection (See, for instance, [IEEESoftware87]).

Taking a quantitative approach, earlier work by Boehm et al. [Boehm78] considered quality characteristics of software, noting that designers' awareness alone improved the quality of the final product. Also supporting a quantitative approach to software quality, Basili and Musa [Basili91] advocate models and metrics of the software engineering process from a management perspective.

Also from a management perspective, Zultner [Zultner91] and Oakland [Oakland89] apply the *Total Quality Management* approach of Deming [Deming86] to software development and maintenance. The application results in guidelines for software managers to know and do (called *fourteen points*), for management to avoid (*seven deadly diseases*), and poor practices that hinder quality and productivity (*fifteen obstacles*). Similarly, albeit less specific to software development, a general management approach to improving productivity has been widely publicized, in the name of *quality (control) circle* (*QC*) which aims to increase efficiency through worker motivation, increased communication among and involvement of employees at all levels within the organization in the decision-making process [Crocker84]. In relation to software quality assurance, some of these and other related work, such as work by Ishikawa, Juran, Sandholm, and Crosby, are surveyed and assessed in [Schulmeyer87].

In the area of information system development, the NATURE project recognizes the crucial role that non-functional or quality requirements play in requirements engineering, and has in its scope the representation of NFRs [Jarke93]. This project attempts to also address teamwork support and the use of repository in dealing with NFRs [Jarke92c], although it is just starting.

General Software Development

In the field of software development, much attention has been given to the generation of efficient implementations from formal specifications, either manually or automatically. For example, [Spivey87] and [Wordsworth87] describe a software development methodology based on a set-theoretic formalism named Z, while [Abrial88] describes a tool (named the B tool) that can serve as proof assistant in establishing that an implementation generated through a sequence of transformations is indeed consistent with a given Z specification.[1] Another project in the same league is the Cornell Program Synthesizer [Teitelman81] which inspired the Nuprl proof development system [Constable86]. Generally, such work focuses on programming-in-the-small applications and is most relevant to the mapping problem from designs to implementations. A fine survey of AI-related work on this problem, which has traditionally been called "automatic programming" can be found in [Barstow87], while [Hayes87] gives an excellent comparative account of several formal specification methodologies.

The Gandalf project at CMU [Haberman86] includes a component called SMILE, which consists of a multi-user software engineering environment. SMILE is intended to offer intelligent assistance to the software developer. However, unlike DAIDA [Jarke92] whose user support comes from a knowledge base

[1] Both Z and the B tool play an important role for the design-to-implementation mapping assistant of DAIDA [Borgida89].

(managed by the GKBMS), the knowledge used by SMILE is hardcoded and cannot be changed or extended easily. [Kaiser87] describes a more recent effort to generalize SMILE by including an *objectbase* and a *model of the software development activity* which represent explicitly some of the hardcoded knowledge of SMILE. This effort is similar is spirit to that of DAIDA. DAIDA differs by focusing on information systems and taking full advantage of its narrow focus in selecting languages, developing tools, and adopting methodologies.

The CHI project [Smith85] [Green86], a descendant of PSI project carried out earlier at Stanford [Green76], is led by Green at Kestrel Institute, and is based on a *wide-spectrum language*[2], called V, that is executable. The starting point here for the software development process is a formal specification consisting of logical assertions which is mapped, via several transformations, to an efficient LISP implementation [Westfold84]. However, the logical assertions that constitute the initial formal specification tend to focus on the behaviour of the software-to-be-built rather than its intended environment and information content and CHI relies on computational notions such as sequencing, compute-versus-store, etc., to make them executable. Thus, CHI, like other projects focusing on programming-in-the-small applications, bypasses the requirements modelling and analysis phase and only addresses the problem of going from a less deterministic description of the software-to-be-built to a completely deterministic (and efficient) one. Also, as pointed out earlier, the CHI project does not offer much guidance on how conceptual system designs are produced. On the other hand, the project does have to its credit a commercial product called REFINE.

As one of the foremost advocates for the need of world modelling in software development, Jackson [Jackson83] proposes a system development method (known by the acronym JSD) which is, in fact, in commercial use. JSD starts with a world model and, through a functional system design, produces an implementation. However, JSD, as other diagrammatic approaches, adopts an *ad hoc* set of modelling features and is often quite informal in the specifications it starts with and produces. The work reported here may be seen as an attempt to go beyond the JSD methodology using concepts from knowledge representation and artificial intelligence.

Although the modelling concepts adopted are not as powerful as ours, the work by Bubenko [Bubenko81] is close to ours in spirit, using layers of formal software specifications as a basis for the definition of a software devel-

[2] A language is wide-spectrum if it offers constructs for specifications ranging from highly abstract and non-procedural to ones that are implementation-oriented [Bauer76].

opment process. The first layer of Bubenko's proposal comprises an abstract world model, called "understanding level", and the second layer comprises a conceptual system design. In order to define model behaviour in non-procedural terms, time is considered as the most essential concept and the world model is described in an extended time perspective rather than in points of procedure invocation. Unlike JSD and the framework reported here, however, Bubenko offers no methodology for mapping the first layer to the second.

ERAE, KAOS, RML/Telos, OOs

Goal-Oriented Methodologies

Systematic goal-oriented approaches have been used in a variety of areas. However, the concern of many of such approaches is not system development, in particular.

Of particular relevance to our work, as briefly mentioned in the previous section, is work in design rationale. In the context of qualitative decision support systems, [Lee91] presents a goal-oriented approach to facilitating decision-making process, which extends an earlier model for representing design rationale [Potts88] by making explicit the goals presupposed by arguments. More specifically, each design issue is treated as a goal to fulfill which is supported, denied or presupposed by various claims. Each claim or its relationship with the associated goal can again be treated as a goal. This leads to a recursive argumentation process, which is captured in term of a dependency representation graph (DRG). In order to support this approach, Lee builds a system, called SYBIL, which consists of a dependency representation language (DRL), four types of services, and a user interface. DRL is used to describe various relationships among claims, and the four types of services are intended for: i) managing such relationships, ii) making use of previous decisions, ii) evaluating the degree of goal satisfaction according to the scheme which each particular decision-maker provides, and iv) allowing alternative decisions with different priorities on the terminal goals. Our work specializes Lee's work to the context of system development with NFRs. When compared to Lee's work, the framework proposed in this book distinguishes goals that are associated with functional requirements from ones that are with NFRs. As will be detailed in Chapter 2, we further categorize the latter types of goals into three types of parameterized goals: NFR, satisficing, and argumentation. This categorization is essential in identifying the right concerns at the right places during system development, and at the same time allowing separation of concerns. This categorization, particularly satisficing goals, makes it possible to use NFRs as criteria for selecting among design alternatives. Our specialization makes it possible to capture different types of NFR-related knowledge, in terms of three types of generic methods corresponding to three types of goals, and later reuse them. When compared to Lee's work, the framework proposed in this book also offers finer-grained link types with a formal semantics, and a built-in, albeit semi-automatic, labelling procedure, and proposes the capture of potential goal synergy or conflict, in terms of correlation rules, and later reuse them. Another benefit of our approach is that our methods, correlation rules, and a labelling procedure make it possible to automate parts of the system development process.

The TI (Transformational Implementation) project [Balzer81] [Fickas85] led by Balzer at USC ISI is one of the foremost automatic programming projects in the world, attempting to develop software through the interaction of system builders and automated assistants. A main theme of the project is the efficient implementation of conceptual system designs constructed in terms of a wide-spectrum executable language, called Gist. Gist is based on state-transitions coupled with the entity-relationship model. Specification refinements are achieved by constraining the allowable state transitions. While treating each design component as a goal to fulfill, the project has generated a set of design alternatives (or methods) and a set of criteria for design decisions that form the backbone for a software development methodology.

KATE is one of several recent projects which address programming-in-the-large. It is led by Fickas at the University of Oregon [Fickas87] [Fickas88] and its scope extends that of the TI project by considering requirements specifications. A fuller specification gradually evolves from an incomplete and inconsistent one, and that is later transformed into a specification that can serve as a starting point for the TI framework. A major difference between our work on one hand and TI and KATE on the other is that Gist is a wide-spectrum executable language while the requirements modelling features in DAIDA project are declarative non-executable knowledge representation language features. Moreover, although Fickas acknowledges the need to consider non-functional requirements in generating conceptual system designs, no concrete framework has been offered as yet for this task. A goal-oriented approach has recently been used also for requirements acquisition [Dardenne91].

In the domain of VLSI high–level synbook, the FAD (Feedback Aided Design) project [Liew90] takes a quantitative approach to improving resource usage by intelligently selecting optimizations and applying them to a previous solution similar to a currently desired solution. This project considers firstly multiple resource goals, such as (quantitative) limitations on time (e.g., time under 2000 units), area (e.g., area under 1000 units), and power (e.g., under 1500 units), and secondly an objective function whose parameters are the resources. In an attempt to improve resource usage of a previous solution, resource goals are, one at a time, decomposed in the manner of a tree; optimization strategies (which will modify a subset of resource components) applicable to the leaves are listed; and the value of the objective function with the optimization strategies are computed. Choosing the better one between the current and the previous value, the designer repeats the improvement process until satisfactory. Although specific contents would be different, the decomposition, optimization strategies, and objective function computation may respectively be comparable to our decomposition, satisficing, and evaluation methods (argumentation is not offered in FAD). Besides the quantitative approach taken here, however, the project does not provide the guidance for detecting potential conflicts among the goals and optimizations or the scheme to capture design rational in the sense of argumentation.

In the area of machine translation, [DiMarco93] and [DiMarco90] presents a scheme for automatically evaluating the translation of one natural language (e.g., English) into another (e.g., French). The evaluation criteria are stylistics such as clarity and dynamicity, which are treated as goals. According to the (stylistic) grammar associated with each goal, each target sentence is categorized into one of three types, each reflecting the bias (or orientation) of the sentence (e.g., a sentence is dynamic, neutral, or static). The work is interesting to us in that it recognizes three types of categories into which the evaluation scheme differentiates each target sentence. As the work's primary concern lies in evaluation rather than translation, it does not address those generation phase issues such as conflict detection and resolution, design rationale maintenance or selection among alternative target constructs.

In the area of computer architecture design, [Agüero87] and [Hooton88] proposes a theory of plausible design (TPD). In this theory, each requirement consists of a combination of functional and non-functional requirements, such as "maximal concurrency", and is treated as a goal to satisfy. The plausibility or satisfiability of the decomposed subgoals (or goal if a terminal node) categorizes each goal into one of four types: validated, assumed, unknown, and refuted. A goal is *validated* if there is a significant evidence for and no evidence against its satisfiability, *assumed* if there is no evidence against its satisfiability, *refuted* if there is some evidence against its satisfiability, and *unknown* when initially introduced. Then, the decomposition of a goal into subgoals, devised by human designers when needed, is guided in the sense that a goal satisfaction of one type can be achieved in terms of its subgoals of the same type. The presence or absence of (counter-) evidence is determined by human designers through simulation, experimentation, literature search, etc. An arbitrary decomposition is also allowed when certain justification is present. The work is of interest as it recognizes the need for distinguishing different types of relationships that can hold between a goal and its subgoals, and subsequently offers an evaluation scheme which is a modification of an ATMS [de Kleer86]. Apart from the lack of a systematic scheme for capturing various design rationale in terms of argumentative structures, this work does not offer (generic) methods or guidelines for detecting potential goal conflicts.

CONTENTS

4	**METHODS AND CORRELATION RULES**	1
4.1	Methods	1
4.2	Decomposition Methods	9
4.3	Operationalization Methods	20
4.4	Argumentation Methods	26
4.5	Correlation Rules	32
4.6	Putting Them All Together: The Goal-Driven Process	42
4.7	Discussion	46
4.8	Related Work for the Framework	47