Feature

Because there is no systematic way in existing use-case approaches to handle nonfunctional requirements, the authors propose an approach to analyze and evaluate use cases with goals and to structure use-case models.

Analyzing User Requirements by Use Cases: A Goal-Driven Approach

Jonathan Lee and Nien-Lin Xue, National Central University

Use-case approaches are increasingly attracting attention in requirements engineering because the user-centered concept is valuable in eliciting, analyzing, and documenting requirements.[1-5] One of the main goals of the requirements engineering process is to get agreement on the views of the involved users,[6] and use cases are a good way to elicit requirements from a user's point of view.

An important advantage of use-case-driven analysis is that it helps manage complexity, since it focuses on one specific usage aspect at a time. Use cases start from the very simple viewpoint that a system is built first and foremost for its users. The approach looks at the interactions of a single category of users at a time, considerably reducing the complexity of requirements determination.[7] However, current use-case approaches are somewhat limited in supporting use-case formalization,[4] and in structuring and managing large use-case models.[8]

0740-7459/99/$10.00 © 1999 IEEE

The standard approaches to improving use cases have two main problems. Although quality issues are often crucial to the success of a software system, no systematic way exists to handle nonfunctional requirements. Also, although interactions between requirements are an important issue in the requirements-acquisition phase,[9,10] there have been no attempts to address them with use cases. Inspired by procedure logic (associating each process with a purpose),[11] we propose an approach to extend use cases with goals, called goal-driven use cases, or GDUC, to alleviate these problems. Our approach

- ♦ uses goals to structure use-case models and derive use cases,
- ♦ differentiates between soft and rigid goals to handle imprecise nonfunctional requirements,
- ♦ embeds goal information in the use cases, and
- ♦ analyzes interactions between requirements by investigating the relationship between goals and use cases.

We chose the meeting schedule problem as an example throughout this article to illustrate GDUC, because the research community has adopted it as a benchmark;[12] the requirements illustrate typical, real system problems; and it can help us address the main challenge of requirements analysis, which is to turn a vague and contradictory mission statement into a detailed specification.[9,12]

Goal-Driven Use Cases

The basic concepts of a use-case approach are *actor* and *use case.* An actor is a specific role played by a system user and represents a category of users that demonstrate similar behaviors when using the system. Use cases describe the way an actor uses the system. A use case has one basic course and several alternative courses. The basic course is the simplest course, the one in which a request is delivered without any difficulty. On the other hand, alternative courses describe variants of the basic course and the errors that can occur.

A use-case model specifies the relationships between use cases, as well as relationships between use cases and actors. Two powerful concepts, *extends* and *uses,* structure and relate use cases. An *extends* relation specifies how one use case may be embedded into another one, extending its functionality. A *uses* relation is used for refinement, extracting similar parts of two or more use cases. In this way, we can describe a similar part once instead of showing a behavior in all use cases.

In Anne Dardenne and her colleagues' approach, goals determine the respective roles of agents in the system and provide a basis for defining which agents should best perform which actions.[13] A goal is a nonoperational objective while a constraint is an operational one; a goal cannot be achieved directly by the application of actions available to some agents. Instead, it is achieved by satisfying the constraints that make it operational. To satisfy these constraints, we may require appropriate restrictions on actions and objects.

Goals representing nonfunctional requirements are rarely accomplished or satisfied.

According to John Mylopoulos and his colleagues, goals represent nonfunctional requirements, design decisions, and arguments in support of or against other goals.[14] Goals representing nonfunctional requirements are rarely accomplished or satisfied in a clear-cut sense. Instead, different design decisions contribute positively or negatively towards a particular goal. They also propose a labeling procedure to determining the degree to which a particular design supports a set of nonfunctional requirements.

In a nutshell, goal-based approaches focus on why systems are constructed and provide the motivation and rationale to justify software requirements. We conceived our approach based on the concept to extend use-case approaches with goals.

Our approach is for requirements engineers to structure and elicit users' requirements, and to analyze and evaluate relationships between requirements. We focus on how to structure and elicit users requirements with a three-step approach to constructing use cases:

1. Identify actors by investigating all possible types of users that interact with the system directly.
2. Identify goals based on a faceted classification scheme.
3. Build use-case models.

Identifying Actors

A person can play several roles and thereby represent several actors, such as computer-system operator or end user.[1] To identify a target system's use cases, we identify the system actors. A good starting

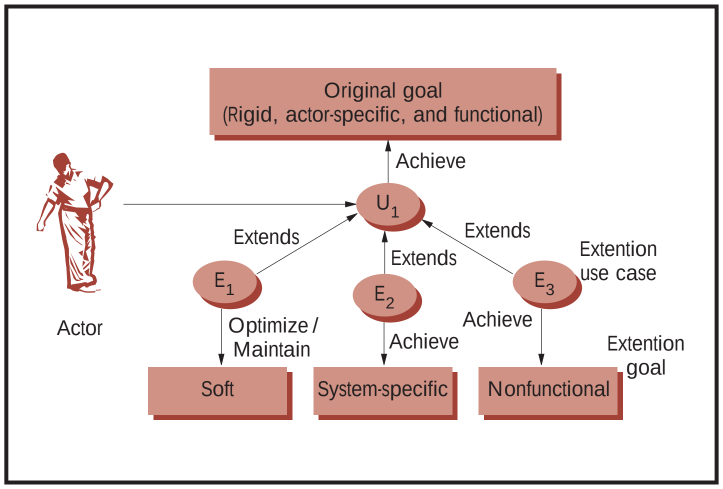


Figure 1. Deriving use cases with goals.

point is to check the system design and who it is supposed to help. For example, the meeting schedule system is mainly designed for an initiator to organize a meeting schedule, so the initiator is an actor. The actors who will use the system are primary actors. Each actor will perform one or more of the main system tasks. Besides these primary actors, secondary actors supervise and maintain the system.

In the meeting schedule system, we identified three actors: an initiator (to initiate a meeting, notify participants, and make decisions if conflicts occur), a participant (to register a meeting and indicate a preference for meeting dates, locations, or equipment), and an operator (to maintain and supervise the system). The initiator and participant are primary actors, whereas the operator is a secondary actor.

IDENTIFYING GOALS

To identify goals from domain descriptions and system requirements, we propose a faceted classification scheme based on our requirement classification scheme.[14] Each goal can be classified with three facets: competence, view, and content.

Competence

The first facet relates to whether a goal is completely or partially satisfied. A rigid goal, which must be completely satisfied, describes a target system's minimum requirement. A soft goal describes a desirable property for a target system and can be partially satisfied. For example, a meeting schedule that is convenient for all attendees completely satisfies its goal, *MaxConvenientSchedule*. However, if the schedule is only convenient to some of the attendees, it is only partially satisfied. A soft goal, therefore, is dependent on the rigid one. That is, a weak relationship exists between a rigid goal and its soft goals.

In our example, the rigid goal *MeetingRequestSatisfied* has two related soft goals, *MaxNumberofParticipants* and *MaxConvenientSchedule*, which are weakly dependent on it, because the information about the number of meeting participants and the meeting's convenience will be meaningless if the meeting request is not satisfied.

View

This facet concerns whether a goal is actor-specific or system-specific. Actor-specific goals are an actor's objectives in using a system; system-specific goals are requirements on services that the system provides. For example, by examining the system description, we found that the initiator has three objectives for the meeting schedule system: to create a meeting, to make the meeting schedule as convenient as possible for the participants, and to maximize the number of meeting participants. Therefore, we identified three actor-specific goals: *MaxNumberofParticipants*, *MeetingRequestSatisfied*, and *MaxConvenientSchedule*.

On the other hand, a system-specific goal considers the properties the system needs to support services for all users as well as those necessary for system operation. Consider this partial system description for the meeting schedule system:

Dynamically replan a meeting to support as much flexibility as possible to take some external constraints into account after proposing a date and location, such as accommodating a more important meeting.

An initiator constructs a meeting, but the system can accommodate a more important meeting. Therefore, we can identify a system-specific goal: *SupportFlexibility*.

Content

We can classify requirements into functional and nonfunctional requirements based on their content.[15] The construction of functional requirements involves modeling the relevant internal states and behavior of both the component and its environment. Nonfunctional requirements usually define the constraints that the product needs to satisfy. Therefore, a goal can be further distinguished based on its content and can be either related to a system's functional aspects or associated with the system's nonfunctional aspect. We achieve a functional goal by performing a sequence of operations. A non-

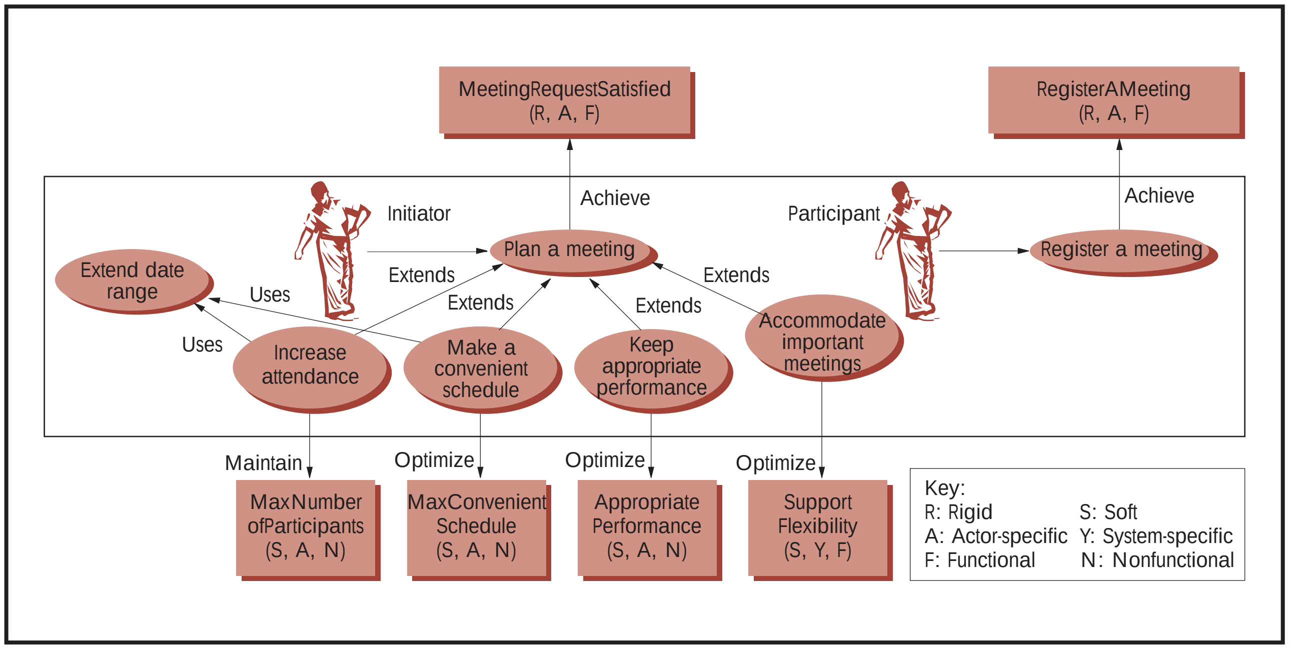

MeetingRequestSatisfied (R, A, F)
RegisterAMeeting (R, A, F)
Achieve
Initiator
Participant
Achieve
Plan a meeting
Register a meeting
Extend date range
Uses
Extends
Extends
Extends
Extends
Uses
Increase attendance
Make a convenient schedule
Keep appropriate performance
Accommodate important meetings
Maintain
Optimize
Optimize
Optimize
MaxNumber ofParticipants (S, A, N)
MaxConvenient Schedule (S, A, N)
Appropriate Performance (S, A, N)
Support Flexibility (S, Y, F)
Key:
R: Rigid
S: Soft
A: Actor-specific
Y: System-specific
F: Functional
N: Nonfunctional

Figure 2. A goal-driven use-case model for a meeting schedule system.

functional goal is defined as constraints to qualify its related functional goal.

In our example, the initiator creates a meeting with a sequence of operations. Therefore, we can define *MeetingRequestSatisfied* as a functional goal. We can also view the goals *MaxNumberofParticipants* and *MaxConvenientSchedule* as constraints for a schedule to satisfy, or as nonfunctional goals.

Building Use-Case Models

In GDUC, we have extended Alistair Cockburn's work by considering several different types of goals to structure a use case and its extensions.[16] Essentially, each use case is viewed as a process associated with a goal that it must achieve, optimize, or maintain (see Figure 2). Building a use-case model involves three steps: identify original use cases to capture minimum requirements, identify extension use cases to construct a more complete model, and refine a use case model to enhance reusability.

Identifying original use cases

We first consider original use cases to guarantee that they can adapt the target system to the minimum requirements. Each original use case in our approach is associated with an actor to describe the process of achieving an original goal, which is rigid, actor-specific, and functional (see Figure 1). Building original use cases by investigating all original goals will make the use-case model satisfy at least all the actors' rigid and functional goals.

The basic course in an original use case is the simplest course—the one in which the goal is delivered without any difficulty. The alternative course encompasses the recovery or failure course, or both. The recovery course describes the process to recover the original goal, whereas the failure course describes what to do if the original goal is not recoverable.

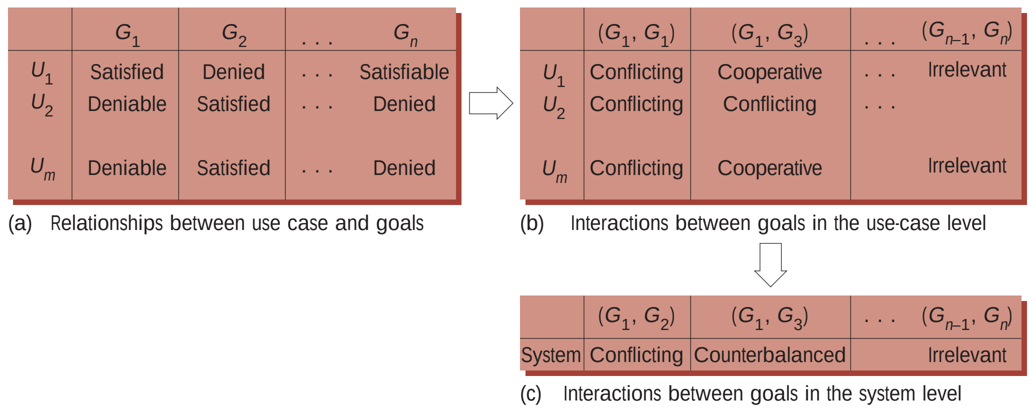
In our example, the use case *plan a meeting* covers the case for an initiator to achieve the goal *MeetingRequestSatisfied*, which is rigid, actor-specific, and functional (see the boxed text, "Sample Use Case," p. 100). The use case starts when an initiator issues a meeting request to the system and lasts until a meeting schedule is generated or canceled. The basic course of *plan a meeting* describes steps to generate a meeting schedule to achieve the goal *MeetingRequestSatisfied*: (1) determining the date range, locations, meeting type, and potential attendees; (2) having participants input their personal agenda; (3) making a schedule based on the given information; and (4) informing all participants of the meeting.

The use case has several alternative courses that may change its flow. For example, there are different ways of recovering the goal *MeetingRequest-Satisfied* when a strong schedule conflict exists.

Note that the problem statements do not indicate when to cancel a meeting, so we will need to elicit this information. The boxed text "Sample Use Case" includes an example of an alternative meeting plan.

Extension use cases

Original use cases are designed to satisfy original goals for modeling users' minimum require-



	G_1	G_2	. . .	G_n
U_1	Satisfied	Denied	. . .	Satisfiable
U_2	Deniable	Satisfied	. . .	Denied
U_m	Deniable	Satisfied	. . .	Denied

(a) Relationships between use case and goals

	(G_1, G_1)	(G_1, G_3)	. . .	(G_{n-1}, G_n)
U_1	Conflicting	Cooperative	. . .	Irrelevant
U_2	Conflicting	Conflicting	. . .	
U_m	Conflicting	Cooperative		Irrelevant

(b) Interactions between goals in the use-case level

	(G_1, G_2)	(G_1, G_3)	. . .	(G_{n-1}, G_n)
System	Conflicting	Counterbalanced		Irrelevant

(c) Interactions between goals in the system level

Figure 3. GDUC's three steps for requirements evaluation.

ments. To extend the model to take into account different types of goals, we create extension use cases.

Optimize or maintain a soft goal

By achieving a rigid goal, we can also satisfy all its related soft goals to some extent. To optimize or maintain the soft goals, we create extension use cases (see Figure 1, the use case E_1). Therefore, the basic course in an extension is to optimize or maintain its soft goal, whereas an alternative course describes what to do if it fails to optimize or maintain the goal.

In our example, satisfying the rigid goal *Meeting-RequestSatisfied* does not guarantee that the meeting is convenient for all participants. To make the schedule as convenient as possible, we create an extension *make a convenient schedule*. If the basic course constraints are not satisfied, the alternative course is to recover the soft goal's optimization. For example, the system may extend the date range or ask participants to add dates to their preference sets.

An extension can also be used to maintain a soft goal. If the basic course of the use case *plan a meeting* goes successfully, all potential participants can attend the meeting. This completely satisfies the soft goal *MaxNumberofParticipants*. However, if an exception or conflict arises, a recovery course may impair the soft goal because some participants may withdraw from the meeting. To maintain the soft goal, we establish an extension *increase attendance*. This extension resolves the conflict by extending the date range or asking participants to remove dates from their exclusive sets. By doing so, it sustains the number of participants and maintains the soft goal.

Achieve a system-specific goal

An extension use case can be created to achieve a system-specific goal (see Figure 1, use case E_2). The original use case *plan a meeting* describes creating a meeting from a personal view, or the initiator's view. The extension use case *accommodate important meetings* extends it to take all actors into account—that is, to achieve a system-specific goal, *SupportFlexibility*.

Achieve a nonfunctional goal

To extend a use-case model to capture nonfunctional requirements, we add extension use cases to achieve a nonfunctional goal (see Figure 1, use case E_3). In this case, an extension use case serves as a constraint to qualify its original use case. In our example, the original use case *plan a meeting* ignores several meeting constraints: *AppropriatePerformance*, *MaxNumberofParticipants,* and *MaxConvenientSchedule*. The basic course of *make a convenient schedule* indicates the soft constraints on a meeting schedule. If the constraints are not satisfied, the alternative course is to recover the soft goal's optimization.

To summarize, an original goal is rigid, actor-specific, and functional, but an extension goal is achieved, optimized, or maintained by an extension use case. An extension goal is weakly dependent on its associated original goal. Satisfying an original goal does not always satisfy its associated extension goals, unless the extension goal is soft.

Refining use-case models

The use-case model as described thus far is sufficient to specify users requirements. However, to further enhance reusability, we need to elaborate the use-case model by looking for common fragments among different use cases and extracting the similar parts into an abstract use case, such as refining the use case model by *uses* relations.

Although both *extends* and *uses* add an additional subsequence into a base sequence, they are essentially different. In an *extends* relation, both extension and original use cases have their corresponding goals to achieve, optimize, or maintain. In a *uses* relationship, an abstract use case enhances reusability and does not have a goal associated with it.

We found that extending a meeting's date range is a common behavior of the use cases *increase attendance* and *make a convenient schedule*, and then this part of behavior is extracted into an abstract use case *extend date ranges* (see Figure 2).

Goal Evaluation

It is important for a good requirements-modeling approach to take real-world entities into account, although the results often contradict one another.[9] Figure 3 shows how GDUC evaluates interactions between goals in three steps:

♦ Analyze the relationships between use cases and goals by investigating the effects on the goals after performing use cases;

♦ Explore the interactions between goals in the use-case level; and

♦ Derive the interactions between goals in the system level.

Relationships between Use Cases and Goals

To better characterize the relationships between use cases and goals, we adopted the proposal from John Mylopoulos and his colleagues on nonfunctional requirements.[14] A goal can be either satisfied or denied, if the goal is completely achieved or ceased, respectively. However, a goal is either satisfiable and deniable if it can be partially satisfied or denied. In addition, a goal is independent if it will not be affected by performing a designated use case.

In GDUC, a use case is designed to achieve, optimize, or maintain its directly associated goals. However, goals not directly associated with the use case can also be affected or cause side effects.

Effects on associated goals

We can satisfy an original goal either by performing the basic course successfully or by recovering the goal from an alternative course (see Figure 4, arrow a). We can also deny the goal under the condition that it is ceased by performing an alternative course.

In GDUC, an extension goal can be either rigid or soft. For a rigid goal, its associated extension use case is designed to achieve and satisfy the goal. On the other hand, an extension use case can be designed to optimize or maintain a soft goal and make it satisfiable (see Figure 4, arrow b). For example, to completely satisfy the soft goal *MaxConvenientSchedule*, we need to successfully perform both use cases *plan a meeting* and *make a convenient schedule*. The role of the extension use case *make a convenient schedule* is to make the soft goal *MaxConvenientSchedule* satisfiable.

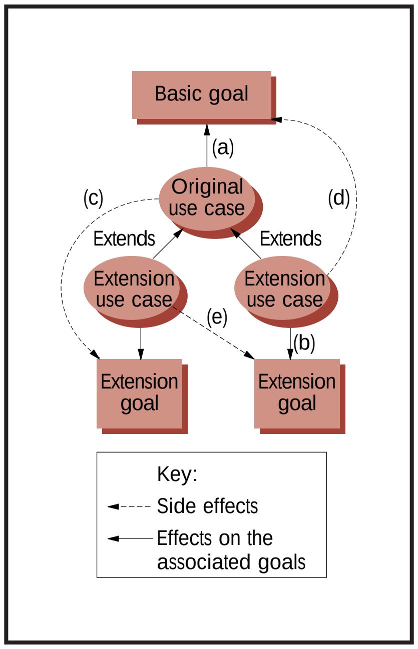


Figure 4. Relationships between use cases and goals.

Side effects

By performing an original use case successfully, we can, to some extent, achieve the extension goals that are directly associated with specific extension use cases (see Figure 4, the arrow c). For example, if the use case *plan a meeting* is successfully performed, the extension goal *MaxConvenientSchedule* is satisfied to a degree.

Generally, an extension use case does not impair the original goal that is directly associated with its original use case, except that the extension is designed to achieve a system-specific goal (see Figure 4, arrow d). In this case, the original goal is denied. For example, the extension use case *accommodate important meetings* may cease the original goal *MeetingRequestSatisfied* under the condition that there is a more important meeting in conflict with it.

An extension use case may also achieve or impair an irrelevant goal associated with other extension use cases (see Figure 4, arrow e). For example, if the extension use case *increase attendance* extends the date range for maintaining its soft goal, another extension goal *MaxConvenientSchedule* may also be satisfiable. A typical example of impairing an irrelevant goal in the meeting schedule system is that performing the extension use case *keep an appropriate performance* may cease the activity to negotiate a convenient schedule. Therefore, the soft goal *MaxConvenientSchedule* is deniable.

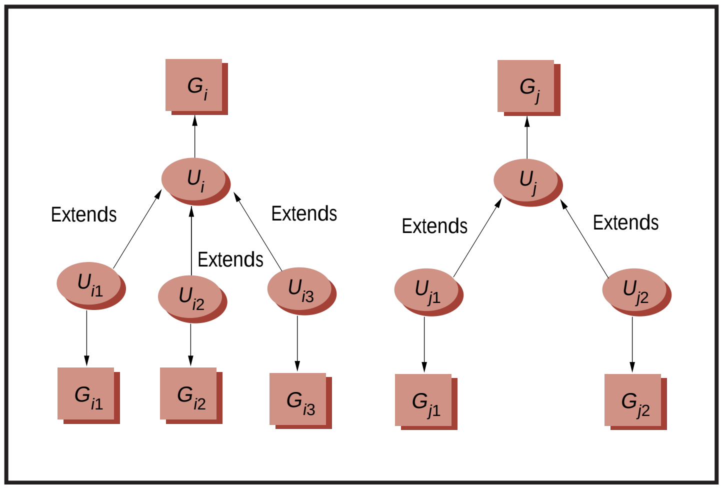


Figure 5. An illustration of a use-case model.

Interactions between goals in the use-case level

In GDUC, relationships between goals exist at two levels: use-case and system. The former is the relationships between goals and specific use cases, and the latter focuses on the overall system.

In the use-case level, two goals conflict with a use case if the satisfaction of one goal increases while the other decreases after the use case is performed. Barry Boehm and Hoh In's system QARCC identifies and diagnoses quality-attribute conflicts.[17] Their approach focuses on domain-independent conflicts involving high-level quality-attribute and architecture-strategy conflicts to achieve generality and scalability. For example, a layered architecture has a positive influence on portability, but has a negative influence on performance. In contrast, our approach deals with the conflicts between requirements.[15] On the other hand, two goals are said to cooperate with each other if both the satisfaction degrees of the goals are either increased (positively cooperative) or decreased (negatively cooperative). The third possibility is that the satisfaction degrees of goals remain unchanged. In this case, the goals are said to be irrelevant.

To obtain interactions between two goals, we need to look at relationships between the use case and those two goals. For example, if the relationships between a use case U_k and goals G_i and G_j are satisfiable and deniable respectively, it means that the satisfaction degree of G_i increases and that of G_j decreases after U_k is performed. G_i and G_j have a conflicting relationship with respect to U_k.

The predicates $cp_{u_k}(G_i, G_j)$ and $cf_{u_k}(G_i, G_j)$ are introduced to describe the relationship between goals G_i and G_j with respect to the use case U_k, where $cp_{u_k}(G_i, G_j)$ is true if G_i and G_j are cooperative with respect to the use case U_k, and $cf_{u_k}(G_i, G_j)$ is true if G_i and G_j are conflicting with respect to U_k. If the goals G_i and G_j are irrelevant with respect to U_k, the predicates $cp_{u_k}(G_i, G_j)$ and $cf_{u_k}(G_i, G_j)$ are both false.

The interactions between the use case *make a convenient schedule* and the goals *MaxConvenient Schedule* and *AppropriatePerformance* are satisfiable and deniable, respectively. Therefore, we can conclude that the two goals are conflicting with respect to the use case *make a convenient schedule*: $cf_{\text{make a convenient schedule}}$*(MaxConvenientSchedule, Appropriate-Performance)=True*.

Interactions between goals in the system level

Interactions between goals in the system level are mainly analyzed on the use-case models where related use cases are amalgamated together. We can derive interactions between goals in the system level based on use-case models and relationships between use cases in the use-case level.

The interaction between the goals G_i and G_j in the system level is denoted as $R_s(G_i,G_j)$, and is defined as a pair of predicates $< cp(G_i,G_j), cf(G_i,G_j) >$, where $cp(G_i,G_j)$ is true if G_i is cooperative with G_j, and $cf(G_i,G_j)$ is true if G_i is conflicting with G_j in the system level. There are four possible relationships between goals in the system level:

- $R_s(G_i,G_j) = < False,False >$: G_i and G_j are irrelevant
- $R_s(G_i,G_j) = < True,False >$: G_i and G_j are cooperative
- $R_s(G_i,G_j) = < False,True >$: G_i and G_j are conflicting
- $R_s(G_i,G_j) = < True,True >$: G_i and G_j are counterbalanced

We describe how to derive interactions between goals in the system level in Figure 5, where G_i and G_j are two original goals and U_i and U_j are their associated use cases, respectively. U_{i1} is an extension use case of U_i, and G_{i1} is its associated extension goal.

Interactions between original goals

We can derive the interactions between any two original goals in the system level by checking the relationships between those two goals with respect to their associated use cases. More specifically, the interactions between G_i and G_j are said to be cooperative if they cooperate with respect to use case U_i or U_j:

$$cp(G_i, G_j) = cp_{U_i}(G_i, G_j) \vee cp_{U_j}(G_i, G_j)$$

Similarly, the interaction between G_i and G_j conflicts if they conflict with the use case U_i or U_j:

$$cf(G_i, G_j) = cf_{U_i}(G_i, G_j) \vee cf_{U_j}(G_i, G_j)$$

By definition, a goal is always cooperative with itself. That is, $R_s(G_i,G_j) = <True, False>$ if $G_i = G_j$.

Interactions between original goals and extension goals

To explore the interaction between an original goal and an extension goal in the system level, we should also consider the original goal on which the extension goal is weakly dependent. In Figure 5, the extension goal G_{j1} is achieved (optimized or maintained) if G_j is satisfied. Thus, we should also consider the interaction between G_i and G_j while investigating the interaction between G_i and G_{j1}. More precisely, let $R_s(G_i,G_{j1}) = <cp(G_i,G_{j1}), cf(G_i,G_{j1})>$, where

$$\begin{aligned} cp(G_i,G_{j1}) &= cp(G_i,G_j) \vee cp_{U_i}(G_i,G_{j1}) \\ &\quad \vee cp_{U_{j1}}(G_i,G_{j1}); \\ cf(G_i,G_{j1}) &= cf(G_i,G_j) \vee cf_{U_i}(G_i,G_{j1}) \\ &\quad \vee cf_{U_{j1}}(G_i,G_{j1}) \end{aligned}$$

We can determine the interaction between the original goal G_{MRS} (or *MeetingRequestSatisfied*) and the extension goal G_{MCS} (or *MaxConvenientSchedule*) by (1) the interaction between the goal G_{MRS} and the original goal G_{MRS} on which G_{MCS} is weakly dependent, (2) the interaction between G_{MRS} and G_{MCS} with respect to the use case *plan a meeting* (U_{PM}), and (3) the interaction between G_{MCS} and G_{MRS} with respect to the use case *make a convenient schedule* (U_{MCS}). That is, $R_s(G_{MRS}, G_{MCS}) = <cp(G_{MRS}, G_{MCS}), cf(G_{MRS}, G_{MCS})>$, where

$$\begin{aligned} cp(G_{MRS},G_{MCS}) &= cp(G_{MRS},G_{MRS}) \\ &\quad \vee cp_{U_{PM}}(G_{MRS},G_{MCS}) \\ &\quad \vee cp_{U_{MCS}}(G_{MRS},G_{MCS}) \\ &= True; \\ cf(G_{MRS},G_{MCS}) &= cf(G_{MRS},G_{MRS}) \\ &\quad \vee cf_{U_{PM}}(G_{MRS},G_{MCS}) \\ &\quad \vee cf_{U_{MCS}}(G_{MRS},G_{MCS}) \\ &= False \end{aligned}$$

Therefore, the goal *MeetingRequestSatisfied* cooperates with the goal *MaxConvenientSchedule*.

Interactions between extension goals

The interaction between two extension goals, $R_s(G_{i1},G_{j1})$ hinges on the system-level interaction between G_{i1} and G_j, the system-level interaction between G_i and G_{j1}, the interaction between G_{i1} and G_{j1} with respect to U_{i1}, and the interaction between G_{i1} and G_{j1} with respect to U_{j1}. That is, $R_s(G_{i1},G_{j1}) = <cp(G_{i1},G_{j1}), cf(G_{i1},G_{j1})>$, where

$$\begin{aligned} cp(G_{i1},G_{j1}) &= cp(G_i,G_{j1}) \vee cp(G_{i1},G_j) \\ &\quad \vee cp_{U_{i1}}(G_{i1},G_{j1}) \vee cp_{U_{j1}}(G_{i1},G_{j1}); \\ cf(G_{i1},G_{j1}) &= cf(G_i,G_{j1}) \vee cf(G_{i1},G_j) \\ &\quad \vee cf_{U_{i1}}(G_{i1},G_{j1}) \vee cf_{U_{j1}}(G_{i1},G_{j1}) \end{aligned}$$

In our example, let $R_s(G_{MCS},G_{AP}) =<cp(G_{MCS},G_{AP}), cf(G_{MCS},G_{AP})>$ be the interaction between the extension goals *MaxConvenientSchedule* (G_{MCS}) and *AppropriatePerformance* (G_{AP}) (G_{KAP} is an abbreviation for the use case *keep appropriate performance*):

$$\begin{aligned} cp(G_{MCS},G_{AP}) &= cp(G_{MCS},G_{MRS}) \\ &\quad \vee cp(G_{AP},G_{MRS}) \\ &\quad \vee cp_{U_{MCS}}(G_{MCS},G_{AP}) \\ &\quad \vee cp_{U_{KAP}}(G_{MCS},G_{AP}) \\ &= False \\ cf(G_{MCS},G_{AP}) &= cf(G_{MCS},G_{MRS}) \\ &\quad \vee cf(G_{AP},G_{MRS}) \\ &\quad \vee cf_{U_{MCS}}(G_{MCS},G_{AP}) \\ &\quad \vee cp_{U_{KAP}}(G_{MCS},G_{AP}) \\ &= True \end{aligned}$$

Therefore, the goals *MaxConvenientSchedule* and *AppropriatePerformance* conflict.

We created GDUC based on the belief that goal information should be captured in the requirements-acquisition phase.[18] In GDUC, goals assist requirements acquisition and modeling in two roles: serving as a structuring mechanism and evaluating requirements.

Our approach offers several benefits:

♦ serving as a structuring mechanism to facilitate the derivation of use-case specifications;

♦ bridging the gap between the domain description and the system requirements—that is, the interactions between functional and non-functional requirements; and

♦ making easy the handling of soft requirements, and the analysis among conflicting requirements.

Sample Use Case: Planning a Meeting

Goal: Determine a meeting date and location (*MeetingRequest-Satisfied*: rigid, actor-specific, and functional).

Basic Course

1. The initiator issues a meeting request.

2. The system automatically fills in the submission date. The initiator fills in the other fields, including the date range, meeting locations, meeting type, and all potential meeting attendants. The initiator also indicates attendees' importance levels.

3. The initiator notifies the potential attendants to input their data. Active participants should fill in the equipment they need. If an attendee is designated an important participant, he is required to fill in his preferred locations. The exclusive sets and preference sets should be contained in the date range.

4. After all participants input their data, the initiator asks the system to make a meeting schedule based on the given information. When making a schedule, the system should consider the proposed meeting date, the stated date range and none of the exclusive set, and find a meeting room available at the selected meeting date. The meeting room should support equipment requirements of all the active participants, and a lower bound should be fixed between the time the meeting date is determined and when the meeting is actually taking place.

5. The system lists all possible meeting schedules that satisfy the criteria. The initiator then chooses one of them and notifies all participants.

Alternative Course

If a strong conflict occurs while generating a meeting, the system will notify the initiator and ask to

- notify a participant to remove a date from his exclusive set,
- propose a participant with low importance level to withdraw from the meeting,
- extend a date range, or
- cancel the meeting.

If none of the proposed locations can meet the equipment requirements while making a meeting schedule, the system should inform the initiator. The initiator can propose other locations or cancel the meeting. In the first case, the initiator first inputs some new locations and then restarts a new round of meeting scheduling. In the second case, the initiator cancels the meeting, and all participants should be informed of the cancellation.

We plan to continue research in two specific areas. Specifically, we will explore the possibility of extending object-oriented models to manage conflicting requirements. After we develop a use-case model with GDUC, we then move to the system-analysis model. As interactions between goals are analyzed in GDUC, the interactions can be utilized for object models to manage conflicting requirements. Additionally, as conflicting requirements are imprecise, we expect object models to possess the capability to model imprecise requirements. A fuzzy object-oriented modeling technique is needed for GDUC to model imprecise and conflicting requirements.

We will also investigate the issues of goal prioritizing using the Analytic Hierarchy Process[19] to support a more accurate requirements model for conflict resolution. The importance of users requirements may vary; some can be critical for the software system's success, while others may merely be adornments. Analyzing the importance of requirements will help construct a software system with high customer satisfaction. ❖

Acknowledgments

The research was supported by National Science Council (Taiwan) under grants NSC87-2213-E-008-024 and NSC88-2213-E-008-006.

References

1. I. Jacobson, *Object-Oriented Software Engineering*, Addison Wesley Longman, Reading, Mass., 1992.
2. J. Rumbaugh, "Getting Started: Using Use Cases to Capture Requirements," *J. Object-Oriented Programming*, Vol. 7, No. 5, Sept. 1994, pp. 8–12.
3. T. Rowlett, "Building an Object Process around Use Cases," *J. Object-Oriented Programming*, Vol. 11, No. 1, Mar./Apr. 1998, pp. 53–58.
4. B. Dano, H. Briand, and F. Barbier, "Progressing towards Object-Oriented Requirements Specifications by Using the Use Case Concept," *Proc. Int'l Conf. Requirements Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 450–456.
5. K.S. Rubin and A. Goldberg, "Object Behavior Analysis," *Comm. ACM*, Vol. 35, No. 9, Sept. 1992, pp. 48–62.
6. K. Pohl, "The Three Dimensions of Requirements Engineering: A Framework and Its Applications," *Information Systems*, Vol. 19, No. 3, 1994, pp. 243–258.
7. P.A. Muller, *Instant UML*, Wrox Press Ltd., Olton, Birmingham, UK, 1997.
8. B. Regnell, M. Andersson, and J. Bergstrand, "A Hierarchical Use Case Model with Graphical Representation," *Proc. IEEE Symp. and Workshop on Engineering of Computer-Based Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 270–277.
9. A. Borgida, S. Greenspan, and J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specification," *Computer*, Apr. 1985, pp. 82–91.
10. A. Finkelstein and R.C. Waters, "Summary of the Requirements Elicitation, Analysis and Formalization Track," *ACM Software Eng. Notes*, Vol. 14, No. 5, 1989, p. 40.
11. M.P. Georgeff and A.L. Lansky, "Procedural knowledge." *Proc. IEEE*, Vol. 74, No. 10, Oct. 1986, pp. 1383–1398.
12. C. Potts, K. Takahashi, and A.I. Anton, "Inquiry-Based Requirements Analysis," *IEEE Software*, Mar. 1994, pp. 21–32.
13. A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming*, Vol. 20, 1993, pp. 3–50.
14. J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *IEEE Trans. Software Eng.*, Vol. 18, No. 6, 1992, pp. 483–497.
15. J. Lee and J.Y. Kuo, "New Approach to Requirements Trade-Off

Analysis for Complex Systems," *IEEE Trans. Knowledge and Data Eng.*, Vol. 10, No. 4, July/Aug. 1998, pp. 551–562.

16. A. Cockburn, "Goals and Use Cases," *J. Object-Oriented Programming*, Vol. 10, No. 7, Sept. 1997, pp. 35–40.
17. B. Boehm and H. In, "Identifying Quality-Requirement Conflicts," *IEEE Software*, Mar. 1996, pp. 25–35.
18. A. van Lamsweerde, R. Darimont, and P. Massonet, *Goal-Directed Elaboration of Requirements for a Meeting Scheduler Problems and Lessons Learnt*, Tech. Report RR-94-10, Universite Catholique de Louvain, Louvain-la-Neuve, Belgium, 1994.
19. J. Karlsson and K. Ryan, "A Cost-Value Approach for Prioritizing Requirements," *IEEE Software*, Sept./Oct. 1997, pp. 67–74.

Readers may contact Lee at National Central University, Department of Computer Science and Information Engineering, Chungli 32054, Taiwan; e-mail yjlee@se01.csie.ncu.edu.tw.

About the Authors

Jonathan Lee is a professor in the Department of Computer Science and Information Engineering at National Central University in Taiwan. His research interests include trade-off requirements, agent-based software engineering, and applications of fuzzy theory to software engineering. He received his PhD in computer science from Texas A&M University. He is a member of the IEEE Computer Society, ACM, and AAAI.

Nien-Lin Xue is a PhD student in the Department of Computer Science and Information Engineering at National Central University in Taiwan. His research interests include requirements engineering, object-oriented methodologies, and the applications of fuzzy logic to software engineering. He received his MS from National Central University and his BS from Soochow University. He is a member of the IEEE Computer Society. He can be reached at nien@se01.csie.ncu.edu.tw.