JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 5, May-June 2004

Goal Directed Analysis with Use Cases

William N. Robinson, Dept. of Computer Information Systems, Georgia State University, U.S.A.
Greg Elofson, Graduate School of Business, University of New Orleans, U.S.A.

Abstract

Use-cases are touted as means to manage the complexity of object-oriented software specification. The UML use-case relationships provide the means to organize use-cases, which in turn, organize use-case requirements. Analysts, unfortunately, have difficulty in determining the scope of a single use-case, as well as defining its elaborations. In response, we define a goal-directed modeling approach based upon foundational definitions for domain property, goal, requirement, and specification. The more formally defined goals guide use-case definition, organization, and enable analyses otherwise unavailable to conventional object-oriented analysis. Goal directed analysis with use-cases helps manage specification complexity.

1 MODELING WITH GOALS

Software specification by use-cases has grown with the popularity of object-oriented software engineering [Weidenhaupt 1998]. Use-cases are now part of every object-oriented analysis method [Regnell 1996], including the popular Unified Modeling Language (UML) and methodology [Fowler 1997]. Analysts, however, have difficulty in decomposing and structuring use-cases. One solution appears to be the use of high-level software goals. Goals can guide use-case development, as well as enable early analysis of software specifications.

Goals

Software controls a small portion of the world. It interacts with its environment. It monitors environmental properties and introduces changes through modification of the logical values or physical effectors that it controls. From these simple observations, we can define four foundational definitions important to the description of software systems, according to van Lamsweerde [van Lamsweerde 2000] and others [Jackson 1995, Parnas 1995].

- A *goal* is a desired property of the environment. For example, "After delivery of an order, the customer shall pay the business."

- A *domain property* is a property that exists naturally in the environment, as it would independent of any software system. For example, “After the production of a perishable product, the product becomes stale.”
- A *requirement* is a special kind of goal that constrains the software behavior. To be a requirement, a goal must satisfy the following three properties: (i) it is described entirely in terms of values monitored by the software; (ii) it constrains only values that are controlled by the software; and (iii) the controlled values are not defined in terms of future monitored values. For example, “Within one day after the delivery of an order, the system shall send an invoice to the order’s customer.”
- A *specification* is special kind of requirement that only references system properties. For example, “The system shall compute product age as the current date minus the product’s production date.”

Most system goals have a form, such as “the system shall do X”, where X is some function that the system shall provide. For example, “the inventory tracking system shall record the inventory level of all products stored in the warehouse.” A slightly more refined view of goals is presented in the following four goal patterns.

- *Achieve* goals require that some property eventually holds; for example, “After the delivery of an order, the system shall send an invoice to the order’s customer.”
- *Cease* goals require that some property eventually stops to hold; for example, “After a past-due account is paid in full, the system shall stop sending invoice reminders to the account’s customer.”
- *Maintain* goals require that some property always holds; for example, “The system shall always record the current inventory level of each inventory product.”
- *Avoid* goals require that some property never holds; for example, “An unauthorized user shall never access any customer account.”

The preceding goal patterns have been formalized [van Lamsweerde 2000], although here they are only presented informally. More generally, Dwyer *et. al*. have analyzed over 500 examples of the kinds of properties that have been used in requirements [Dwyer 1999]. They found that nearly all conformed to eight temporal patterns, which a refinements of the preceding patterns [Dwyer 1999].

Goal-Oriented Modeling

Goals are used in a variety of ways to analyze software systems[Kavakli 2000]. Perhaps, van Lamsweerde says it best[van Lamsweerde 2000]:

> *Goals drive the elaboration of requirements to support them; [Dardenne 1991, Ross 1977, Rubin 1992] they provide a completeness criterion for the requirements specification —the specification is complete if all stated goals are met by the specification[Yue 1987]; they provide a rationale for requirements —a requirement exists because of some underlying goal which provides a base for it [Dardenne 1991, Sommerville 1997]; goals represent the roots for detecting conflicts among requirements and for resolving them eventually [Robinson 1989, van Lamsweerde 1998]; goals are generally more stable than the requirements to achieve them [Anton*

1994]. In short, requirements "implement" goals much the same way as programs implement design specifications.

Although goals are widely recognized as important, their use in object-oriented modeling is rare—particularly, with the UML methodology.

Cockburn is often cited as having introduced goals to object-oriented analysis [Cockburn 1997, Cockburn 1997]. He defines use-cases to satisfy goals: “All the interactions relate to the same goal. … The goal is a strategic goal with respect to the system.” He sees five opportunities for goals: (1) attach non-functional requirements to goals, (2) track the project by goals, (3) get subtle requirements from goal failures, (4) use goals with responsibility-based design, and (5) match user goals to operational concepts. More recently, Bock shows how goals can assist in choosing parameters from the object model [Bock 2000, Bock 2001].

Sometimes goals are called features. For example, Leffingwell defines a feature as, “a service the system provides to fulfill one or more stakeholder needs.”—p. 89 [Leffingwell 2000]. According to Leffingwell, software satisfies requirements, which satisfy use-cases, which satisfy features, which satisfy user needs. Thus, analysts use different documents to describe different levels of system abstraction. Although people may not agree on the term—goal, feature, or softgoals [Gross 2001]—most agree that goals provide a target for the more refined software specification that follows. However, no one has provided a method showing how to derive UML use-case specifications from system goals.

2 A GOAL-ORIENTED METHOD

We define a method for deriving UML specifications from goals. The method is a synthesis of common UML methods, such as the Rational Unified Process [Kruchten 2000], and goal-oriented requirements analysis methods, such as KAOS [Dardenne 1993]. The method consists of five activities:

1. *Elicit the system context.* Information about the proposed system, and its context, are acquired through interviews, document gathering, observation, *etc*.
2. *Define the system goals*. Based on the system context, an analyst defines the system goals.
3. *Derive requirements.* Goals are refined to the requirements level.
4. *Derive use-cases*. Organizational, system, and low-level use-cases are derived from the requirements.
5. *Derive UML models*. Other UML models, such a class and sequence diagrams, are derived from the use-cases or requirements.

Elicitation is common to all systems analysis methods. Defining goals and deriving requirements is common to goal-oriented methods. Finally, defining use-cases at varied abstraction levels and deriving their associated models is common to object-oriented methods.

Our goal-driven object-oriented approach to analysis provides direction to what otherwise has been a complex process. In fact, many methods provide more alternative activities than specific directions. Consequently, analysts become a drift is a sea of notations and possibilities.

Adding goals to UML method of analysis provides the following benefits.

- *Abstraction*. Goals provide high-level, functional and non-functional, understandable descriptions of what the system shall do, without the complexity of describing how the system works [van Lamsweerde 2001].
- *Direction*. Goals provide analysts with a checklist of activities to complete [Sommerville 1997, Yue 1987].
- *Traceability*. Goals provide a bridge linking stakeholder requests to system specification [Robinson 1990, Robinson 1998].
- *Analysis*. Goals provide a means to analyze the system prior to its construction. Such analysis is important, and includes: conflict analysis [Robinson 1994, van Lamsweerde 1998] and coverage analysis [Yue 1987].

Next, we present the activities of defining system goals to deriving use-cases (preceding steps 2 – 4); goals and their relationship to UML is the major emphasis. The presentation draws on two systems development exemplars [Feather 1997]: (1) the elevator problem, and (2) a common order processing system.

3 DEFINING SYSTEM GOALS AND REQUIREMENTS

Analysts define desired properties of the environment, or goals, based on stakeholder needs. As environmental statements, goals do not explicitly constrain software behavior; that is the job of requirements. Analysts refine goals by adding details, which typically constrain the software. Thus, requirements can be derived from goals by refinement.

Analysts structure goals according to how they relate to each other. Structuring is important when there are many goals. Perhaps, systems with a small number of goals, say 25, may simply provide a goal list. Most systems, however, must provide a hierarchical structuring of goals.

To define a goal hierarchy, an analyst needs at least one initial goal and two questions: how? and why? Some initial goals can be obtained through interviews, observations, and the review of existing documents and systems. Next, the analyst selects a goal and asks: “How can this goal be satisfied?” and “Why is this a system goal?” How questions are answered by refining the goal into subgoals; this expands the hierarchy downward by introducing goals that are more specialized. Generally, a goal G, may be satisfied by the conjunction of subgoals: G_1 and G_2 and… G_n. Of course, there may be more than one way to satisfy a goal. Thus, a goal G, may be satisfied by the disjunction of subgoals: G_1 or G_2 or… G_n. Answering the why question expands the hierarchy in the opposite direction, by introducing goals that are more abstract.

Figure 1 illustrates a hierarchical goal structure. . The most abstract goal, G_1, is shown at the top, while the most specific goals, $G_{8.1}$ and $G_{9.1}$, are shown at the bottom. Goals G_8 and G_9 are shown as two alternatives means to satisfy goal $G_{1.2.1}$. Therefore, we describe the refinement of goal $G_{1.2.1}$, as an *or*-refinement. In contrast, the refinement of goal G_1 is shown as an *and*-refinement: all subgoals of an and-refinement must be satisfied as a means to satisfy the goal.

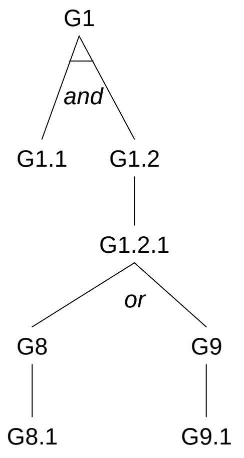
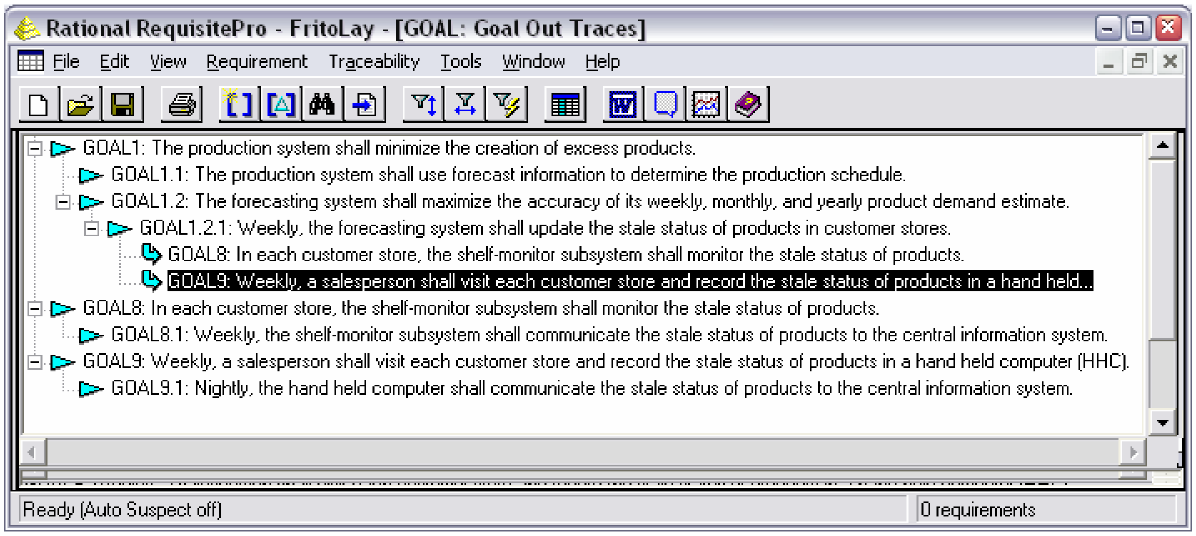


Figure 1 Goal hierarchy.

Figure 2 shows the hierarchical goal structure of Figure 1 as represented in RequisitePro™. Indentation and hierarchical numbering capture the same information, while allowing for a more command, and practical, textual presentation.


Rational RequisitePro - FritoLay - [GOAL: Goal Out Traces]
File Edit View Requirement Traceability Tools Window Help

GOAL1: The production system shall minimize the creation of excess products.
    GOAL1.1: The production system shall use forecast information to determine the production schedule.
    GOAL1.2: The forecasting system shall maximize the accuracy of its weekly, monthly, and yearly product demand estimate.
        GOAL1.2.1: Weekly, the forecasting system shall update the stale status of products in customer stores.
            GOAL8: In each customer store, the shelf-monitor subsystem shall monitor the stale status of products.
            GOAL9: Weekly, a salesperson shall visit each customer store and record the stale status of products in a hand held...
GOAL8: In each customer store, the shelf-monitor subsystem shall monitor the stale status of products.
    GOAL8.1: Weekly, the shelf-monitor subsystem shall communicate the stale status of products to the central information system.
GOAL9: Weekly, a salesperson shall visit each customer store and record the stale status of products in a hand held computer (HHC).
    GOAL9.1: Nightly, the hand held computer shall communicate the stale status of products to the central information system.

Ready (Auto Suspect off) | 0 requirements

Figure 2 A RequisitePro view of the goal hierarchy.

Refinement patterns

An analyst creates a goal hierarchy by refining goals. A goal is refined by adding more specific details. As an illustration, consider providing a friend with directions to your house. First, you might suggest an overview: “From your location, you will need to get on a highway, drive south, go through some business and residential streets, and then you will arrive.” Next, you might refine your description by describing the details of the highway, the intermediate streets, and finally your address. Just as milestones can aid driving, they can aid the refinement of goals.

Table I presents the two basic patterns that analyst use to generate a goal hierarchy. Disjunction is the first. It simply specifies alternatives means to satisfy a goal. Conjunction is the second. It refines the description of a goal. The more specific details provided expand the goal hierarchy toward the operational descriptions needed by software designers and programmers.

Two refinement patterns are often used: milestone and case-based [Darimont 1996]. The *milestone refinement pattern* decomposes achievement into a set of intermediate steps, the sum of which adds up to satisfy the overall goal. The *case-based refinement pattern* decomposes achievement into a set of cases, which add up to satisfy the overall goal. As with all refinement patterns, the sum of the *and*-subgoals must satisfy the goal.

Table I Patterns for elaborating a goal hierarchy.

	Type	Definition	Example
Disjunction (*or*)	*Basic*	$G_1 \leftarrow G_{1.1}$ *or* $G_{1.2}$	After the elevator arrives at a floor, the floor display shall indicate arrival. ← After the elevator arrives at a floor, the floor display shall sound a chime. *or* After the elevator arrives at a floor, the floor display shall floor number shall light.
Conjunction (*and*)	*Basic*	$G_1 \leftarrow G_{1.1}$ *and* $G_{1.2}$	After the elevator arrives at a floor, the display lights shall be updated. ← After the elevator arrives at a floor, the call light shall become unlit. *and* After the elevator arrives at a floor, the floor number light shall light.
	Milestone	After P then Q ← After P then M *and* After M then Q	After the elevator call button is pressed, the call button shall light. ← After the elevator call button is pressed, then the controller shall be notified of the button press. *and* After the controller is notified of a button press, then the call button shall light.
	Case-based	If P then Q ← If C_1 then Q *and* If C_2 then Q *and* P implies C_1 *or* C_2	An elevator shall not shutdown while there are pending requests. ← An elevator shall not shutdown while there are pending call requests. *and* An elevator shall not shutdown while there are pending open door requests. *and* An elevator shall not shutdown while there are pending close door requests. *and* Pending requests means call, open door, or close door.

When to stop asking how?

By refining goals, an analyst creates detailed descriptions, perhaps even approaching program definitions. Consider starting with goal, G_1 and then refining it to $G_{1.1}$ and $G_{1.1.1}$, *etc*. When should refinement stop? When should an analyst stop asking *how*? To answer this question, it is helpful to have a deeper understanding of requirements.

Ideally, requirements are a minimal set of descriptions that constrain the system behavior as a means to bring about desired properties of the environment. Domain properties need only be included as necessary. For example, as part of goal refinement: "Given that goal G_1 can be satisfied by first $G_{1.1}$ and then $G_{1.2}$, we need only implement $G_{1.2}$ because $G_{1.1}$ is satisfied by the environment through domain property, P_1." Similarly, requirements need only be included as necessary to describe the system's interactions with the environment.

Unfortunately, it can be difficult for analysts to recognize when they are describing unnecessary domain properties and system details. For example, given a series of goal refinements, $G_1 \leftarrow G_{1.1} \ldots \leftarrow G_{1.1.1.1.n}$, at what point does the n^{th} subgoal become an unnecessary implementation detail rather than a system requirement?

A *requirement* simultaneously describes the environment and the system. In so doing, it specifies a portion of the system and the domain properties on which it depends. A requirement derived from many refinement steps probably lacks references to the environment. In fact, a description that only references system properties is a special kind of requirement, called a *specification*.

The term *requirement* has been used in a variety of ways. We adopt Jackson's approach, which he so eloquently describes in his book, Software requirements & specifications: a lexicon of practice, principles, and prejudices [Jackson 1995]. (Jackson refers to an elevator as a lift, and the system as the machine.)

Requirements are about the phenomena of the application domain, not about the machine. To describe them exactly, we describe the required relationships among the phenomena of the problem context. A lift passenger who wants to travel from the third to the seventh floor presses the Up button at the third floor.

The light beside the button must then be lit, if it was not lit before. The lift must arrive reasonably soon, traveling in an upwards direction. The direction of travel is indicated by an arrow illuminated when the lift arrives. The doors must open, and stay open long enough for the passenger to enter the lift. The doors must never be open except when the lift is stationary at the floor. There's nothing here about the machine that will control the lift.

But the machine can ensure that these requirements are satisfied because it shares some phenomenon with the application domain: they have some events or states in common. When a shared event happens, it happens in both; a shared state, with its changes of value, is visible and both. For example, pressing the lift button is an event common to the application domain and the machine that controls the lift. To the passenger the event is 'Hit the Up button on the third floor'. The machine may see it as *'input signal on line 3U'. But they are both participating in the same event. Another shared event is the activation of the lift winding motor. The event 'turn winding motor on' in the problem context is the same event as 'output signal on line M+' in the machine. […]*

But not all the phenomenon of the problem context are shared with the machine. For example, the movement of the lift when it is traveling between floors is not shared. The machine has no direct indication of the lift travel until it reaches the next sensor. Nor are the entry and exit of each passenger shared events. The machine has no way of knowing that the passenger who pushed the request button to travel to floor 4 actually got out at floor 2.

In general, that opens up a gap between the customer's requirements and what the machine can achieve directly, because the customer's requirements aren't limited to the phenomena shared with the machine. –pp. 169 – 170 [Jackson 1995].

Figure 3 illustrates required behaviors as the intersection between environmental behaviors and implementable behaviors [Jackson 1995]. The system interacts with a portion of the world, which exhibits the environmental behaviors, as represented in domain properties. Implementable behaviors are executed by the system. A specification describes how the system produces its behaviors. A requirement refers to properties of both the environment and the system. A domain property only refers to properties of the environment. A system specification only refers to properties of the system.

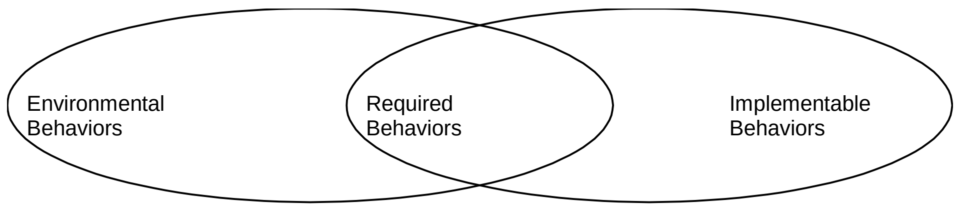


Figure 3. Requirements as the boundary between environment behaviors and implementable behaviors.

For an analyst, goal refinement should stop when the goal descriptions no longer refer to domain properties. After that point, development moves from the analysis phase to the design phase. Of course, a designer may wish to further refine the goals as a means to describe the inner workings of the system.

One cannot distinguish between a requirement and a specification without knowing what the system is intended to do. Passenger movement is an elevator system goal. Consequently, elevator controller descriptions are requirements. However, winding motor descriptions are implementation details because they do not refer directly to passenger domain properties; thus, winding motor descriptions are specifications in this context. Their refinement begins in the later phase of system design.

What if we work at the elevator winding motor factory? Our goal is to produce efficient, silent, and smooth winding motors. For us, winding motor descriptions are requirements, because they refer to our system as well as our domain properties. “The motor shall act as a brake against gravity to smooth the descent of an elevator,” is a requirement for employees of a winding motor company.

Goals, requirements, and specifications are similar. As presented in the introduction, a *goal* is a desired property of the environment. A *requirement* is a special kind of goal that has certain restrictions on use of monitored and controlled values. A *specification* is more restricted, in that it only refers to system properties. Analysts use goals to help decide, for the system at hand, if a description is a requirement or if it is a specification. This is important, because specification work can be set aside, until the requirements analysis is satisfactory. Thus, an analyst can say, “If I refine goal $G_{1.1.1.1}$., it will be the start of specifying the system. I had better finish my requirements before I begin the specification phase. So, I’ll check the other goals to see if they are sufficiently refined.”

Why ask why?

By asking Why questions, an analyst can derive rationale for system goals. For example, an analyst may be presented with the goal, “The elevator controller shall open and close the elevator doors.” “Why?”, the analyst asks. For an elevator controller, it may appear obvious. An elevator controller exists to control the doors and move the elevator between floors. Still, why do we have elevator controller software?

By asking Why questions of the elevator system, analyst are led to consider the cost of the elevator controller. Earlier, elevator operator labor was relatively inexpensive and elevator control software was nonexistent. Now, elevator control software is cheaper than operator labor, when amortized over the life of the elevator. Consequently, when the decision about the elevator controller is revisited in light of the software solution, the software solution appears best. Thus, high-level rational guides decisions among lower-level choices.

4 DERIVING USE-CASES

In UML, a use-case “describes a sequence of actions a system performs that yields a result of value to a particular actor”—p.491 [Leffingwell 2000]. Use-cases can describe a system at different levels of abstraction [Regnell 1996]. We recognize three common use-case types, based on their actors and use of specification statements [Cockburn 1997, Larman 2001]:

- *Organizational* use-cases include actors from multiple organizations, with a focus on documenting the information flow among organizations. Each statement in the use-case is a goal or requirement, as defined in the introduction. An inter-organizational workflow use-case is a typical example.
- *Task* use-cases include actors from only a single organization, with a focus on designing the information processing needed to provide value to the actors. Each

statement in the use-case is either a goal or requirement. A user interface use-case is a typical example.

- *Low-level* use-cases support the definition of task use-cases as a means to decompose or organize use-cases; use-case statements are specifications in that they only reference system properties, and do not reference domain properties. A system service use-case, such as data persistence, is typical example.

We consider use-cases based on goal statements as *abstract*, and use-cases based on requirements or specifications are *concrete*.

Analysts derive use-cases from the goal hierarchy. Consider each node:

- If it has subgoals, then an abstract use-case can be defined with the subgoals as use-case steps.
- If it has subrequirements, then a concrete use-case can be defined with the subrequirements as use-case steps.
- If it is a leaf requirement, then a low-level use-case can be defined using specification statements.

A node with an *and*-refinement has each subnode (goal or requirement) as a properly ordered step in the use-case. A node with an *or*-refinement can either: (1) include only one of the nodes, or (2) include each subnode, along with a condition of its application, thereby, defining alternative use-case paths. By considering the children, and other ancestors, of the subnodes, further use-case details can be added directly to the use-case, or through separate use-case extensions.

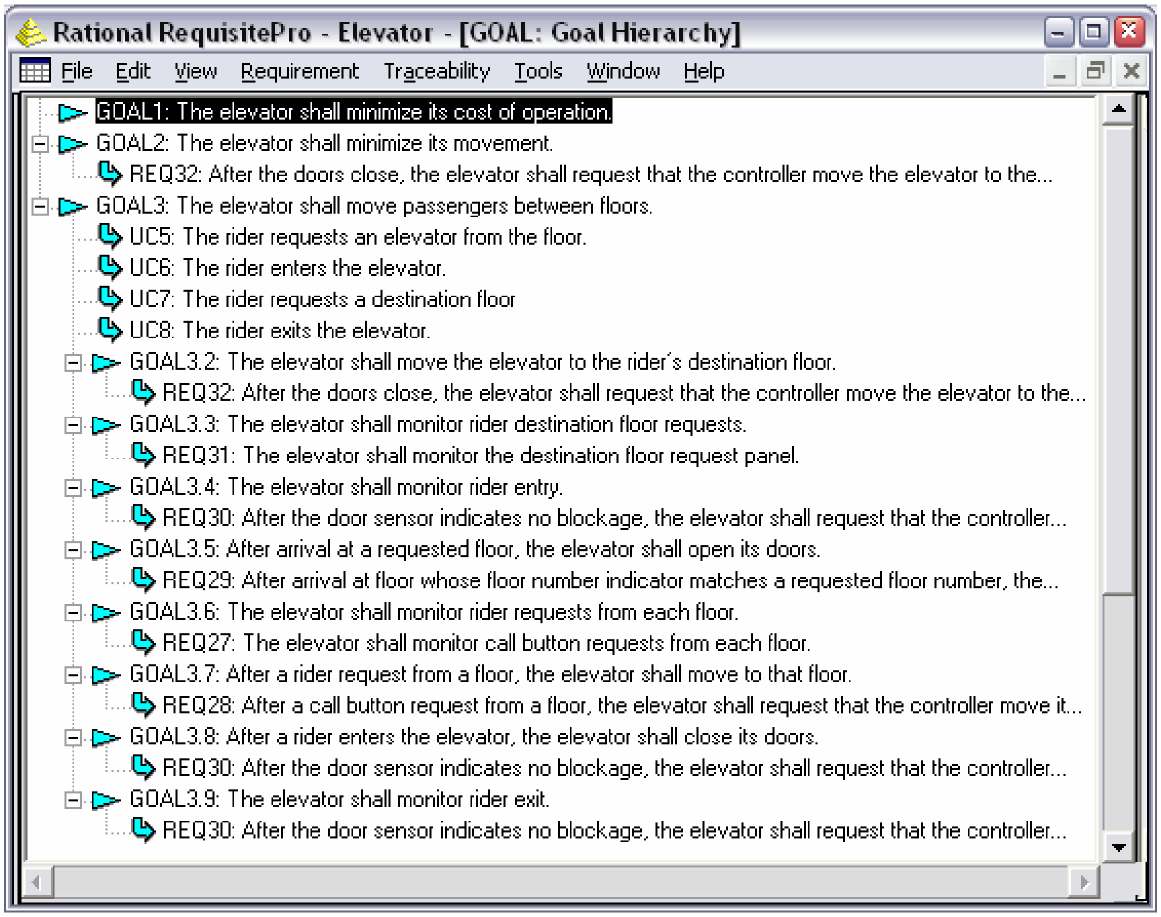
Consider the following statement: "S_1: An unauthorized user shall never access any customer account." It is a goal, because the software system is not constrained. Now, consider the following requirement that is part of the goal's refinement: "S_2: The system shall authenticate each user identity with an external authentication service." An organizational use-case can describe the interactions among the user, software system, and the authentication server. Moreover, based on two lower-level nodes, two task use-cases can further refine the description of how the system will manage the interactions: (1) a user login task use-case, and (2) an authentication request task use-case. Of course, both of these task use-cases can be refined into low-level use-cases using the UML uses and extends relationships.

5 ELEVATOR REQUIREMENTS

Consider the common exemplar of specifying an elevator controller. The following defines three high-level goals.

```
The elevator shall minimize its cost of operation.
The elevator shall minimize its movement.
The elevator shall move passengers between floors.
```

The third goal can be refined, using milestones, to eight subgoals; they are shown as `GOAL3.2` to `GOAL3.9` in Figure 4. They also appear as the system statements in the abstract use-case of Table 2.


Rational RequisitePro - Elevator - [GOAL: Goal Hierarchy]

File Edit View Requirement Traceability Tools Window Help

```
GOAL1: The elevator shall minimize its cost of operation.
GOAL2: The elevator shall minimize its movement.
    REQ32: After the doors close, the elevator shall request that the controller move the elevator to the...
GOAL3: The elevator shall move passengers between floors.
    UC5: The rider requests an elevator from the floor.
    UC6: The rider enters the elevator.
    UC7: The rider requests a destination floor
    UC8: The rider exits the elevator.
    GOAL3.2: The elevator shall move the elevator to the rider's destination floor.
        REQ32: After the doors close, the elevator shall request that the controller move the elevator to the...
    GOAL3.3: The elevator shall monitor rider destination floor requests.
        REQ31: The elevator shall monitor the destination floor request panel.
    GOAL3.4: The elevator shall monitor rider entry.
        REQ30: After the door sensor indicates no blockage, the elevator shall request that the controller...
    GOAL3.5: After arrival at a requested floor, the elevator shall open its doors.
        REQ29: After arrival at floor whose floor number indicator matches a requested floor number, the...
    GOAL3.6: The elevator shall monitor rider requests from each floor.
        REQ27: The elevator shall monitor call button requests from each floor.
    GOAL3.7: After a rider request from a floor, the elevator shall move to that floor.
        REQ28: After a call button request from a floor, the elevator shall request that the controller move it...
    GOAL3.8: After a rider enters the elevator, the elevator shall close its doors.
        REQ30: After the door sensor indicates no blockage, the elevator shall request that the controller...
    GOAL3.9: The elevator shall monitor rider exit.
        REQ30: After the door sensor indicates no blockage, the elevator shall request that the controller...
```

Figure 4. The elevator goal hierarchy.

The preceding goals are system goals, whereas most use-cases are derived from actor goals [Cockburn 1997]. System goals derive a systems view that is compatible with Jackson's view of requirements: system specification is the focus because users cannot be directly constrained. Nevertheless, one can use actor goals to derive the elevator system. In doing so, the actor actions imply the system actions. Here, we show how the system actions imply the actor actions.

To derive a use-case from `GOAL3`, an analyst places each subgoal, in its proper order, as a system action. Next, the actor actions are defined for each system action requiring input or accepting output, for example, the rider's elevator request, in step 3, provides the input for the system statement in step 4. The resulting abstract use-case is shown in Table 2.

Table 2. An abstract task use-case for riding an elevator.

Rider (Actor)	Elevator (System)
1.	2. The elevator shall monitor rider requests from each floor.
3. The rider requests an elevator from the floor.	4. After a rider request from a floor, the elevator shall move to that floor.
5.	6. After arrival at a requested floor, the elevator shall open its doors.
7.	8. The elevator shall monitor rider entry.
9. The rider enters the elevator.	10. After a rider enters the elevator, the elevator shall close its doors.
11.	12. The elevator shall monitor rider destination floor requests.
13. The rider requests a destination floor .	14. The elevator shall move the elevator to the rider's destination floor.
15.	16. After arrival at a requested floor, the elevator shall open its doors.
17.	18. The elevator shall monitor rider exit.
19. The rider exits the elevator.	20.

The system statements of Table 2 are goals rather than requirements because they not realizable [Letier 2002]. In particular, they do not describe monitored and controlled variables. For example, how can the elevator monitor rider requests (GOAL3.6)? Should riders beam their requests from their hand-held devices? (Implicitly, a set of domain properties defines the monitored and controlled variables.) Refinement of the goals into requirements provides for a concrete use-case, as shown in Table 3.

Table 3. A concrete task use-case for riding an elevator.

Rider (Actor)	Elevator (System)
1.	2. The elevator shall monitor call button requests from each floor.
3. The rider presses the call button .	4. After a call button request from a floor, the elevator shall request that the controller move it to that floor.
5.	6. After arrival at floor whose floor number indicator matches a requested floor number, the elevator shall request that the controller open the doors.
7. The rider enters the elevator.	8. After the door sensor indicates no blockage, the elevator shall request that the controller close the doors.
9.	10. The elevator shall monitor the destination floor request panel.
11. The rider shall press the destination button.	12. After the doors close, the elevator shall request that the controller move the elevator to the closest requested destination floor.
13.	14. After arrival at floor whose floor number indicator matches a requested floor number, the elevator shall request that the controller open the doors.
15. The rider exits the elevator.	16.

The system statements of Table 3 are requirements refined from the goals of Table 2. The goal hierarchy of Figure 4 shows all the relationships; it shows: (1) goal refinement, (2) derived requirements, and (3) use-case actor actions associated with goals (e.g., UC5). Although not shown in Table 3, some actor actions have been refined from their abstract counterparts in Table 2; for example, UC5.1 indicates that the rider presses the monitored call button, rather than placing an abstract request.

The definitions for goal, requirement, and specification guide the definition of the goal hierarchy and use-cases. Goals provide rationale and structure for the requirements that provide sufficient details for use-case definition. Their definitions, along with actor differentiation, clearly sort use-cases into organizational, task, or low-level. Alternative classifications, such as essential and real, are more subjective [Larman 2001].

Analysts derive UML models from requirements, be they free form requirements, or use-case requirements. A simple approach, for example, is to define a UML class for each noun occurring in a requirement; similarly, requirement actions become class operations [Rumbaugh 1991]. Further analysis may reveal that the derived definitions need to be combined—because of synonyms, for example. The goal hierarchy, like use-cases, consist of text statements. Therefore, analysts can also mine them for candidate classes and operations. Formalized goals enable the automated derivation of classes and operations [Letier 2002].

6 DISCUSSION

We have applied the goal directed UML modeling approach in industrial and university settings. We offer the following as *lessons learned*, rather than conclusions, because of their anecdotal nature.

- Analysts are quick to grasp the foundational definitions when they a given a number of examples—good and bad.
- Analysts find it natural to generate goal hierarchies using How and Why questions.
- Analysts can quickly generate good use-cases from the goal hierarchy.

Our future research plans include a controlled experiment to test the effectiveness of the approach. In particular, we believe that analysts can produce better use-cases in shorter time using the goal directed approach, than they can by using classic use-case guidance, or no guidance. The theory of goal directed problem solving provides support: it applies to object-oriented analysis [Purao 2002], programming [Soloway 1984], as well as general problem solving [Newell 1972]. We believe that the goal directed approach to object-oriented specification is easily learned and effectively applied, because of the familiarity and power of the underlying problem solving approach.

Generally, analysts have difficulty overcoming the complexity of object-oriented methods. Use-cases are considered to be the answer. Of course, that leads to the question of how use-cases are defined and organized as a means of reducing complexity. Use-case types, such as essential and real, are supposed to guide analyst; however, their definitions typically generate more questions than answers.

The definitions for domain property, goal, requirement, and specification provide a foundation upon which complexity can be conquered. Based on the form of their descriptions, analysts know if their descriptions are defining the domain, domain changes, software requirements, or the internals of software. The definitions guide the development of the goal hierarchy. Guided by the goal hierarchy, analysts derive use-cases can defined. (Use-cases based on goals can describe ideal behaviors. However, goals must be realizable before a software can be defined; for example, software values cannot be defined in terms values to be observed in the future [Letier 2002].) Use-cases are defined at the leaves of the goal hierarchy, thus, they are partitioned, thereby partitioning them into functional groups, naturally. Overall, goal directed analysis with use-cases reduces complexity and guides analysis.

ACKNOWLEGEMENTS

This paper has benefited greatly from reviews provided by Greg Elofson and Sandeep Purao. Thanks! Of course, any remaining errors are solely the responsibility of the author.

REFERENCES

[Anton 1994] A. I. Anton, W. M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering," Proceedings of Conference on Advanced Information Systems Engineering (CAISE'94), LNCS 811, 1994.

[Bock 2000] C. Bock, "Goal-driven Modeling," *Journal of Object-Oriented Programming*, vol. 13, pp. 48–53, 2000.

[Bock 2001] C. Bock, "Goal-driven Modeling, Part II," *Journal of Object-Oriented Programming*, vol. 14, 2001.

[Cockburn 1997] A. Cockburn, "Goals and Use Cases," *Journal of Object-Oriented Programming*, vol. 10, pp. 35-40, 1997.

[Cockburn 1997] A. Cockburn, "Using Goal-Based Use Cases," *Journal of Object-Oriented Programming*, vol. 10, pp. 56-62, 1997.

[Dardenne 1991] A. Dardenne, S. Fickas, and A. v. Lamsweerde, "Goal-Directed Concept Acquisition in Requirements Elicitation," Proceeings of the Sith Intl. Workshop on Software Specification and Design (IWSSD-6), Como, 1991.

[Dardenne 1993] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computing Programming*, vol. 20, pp. 3-50, 1993.

[Darimont 1996] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," Fourth Symposium on the Foundations of Software Engineering, San Francisco, CA, 1996.

[Dwyer 1999] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," Twenty-First International Conference on Software Engineering, Los Angeles, 1999.

[Feather 1997] M. S. Feather, Fickas, S., van Lamsweerde, A., "Requirements and Specification Exemplars," *Automated Software Engineering*, vol. 4, pp. 419-438, 1997.

[Fowler 1997] M. Fowler and K. Scott, *UML Distilled - Applying the Standard Object Modeling Language*. Readings, MA: Addison-Wesley, 1997.

[Gross 2001] D. Gross and E. Yu, "From Non-Functional Requirements to Design through Patterns," *Requirements Engineering. Springer-Verlag*, vol. 6, pp. 18-36, 2001.

[Jackson 1995] M. J. Jackson, *Software requirements & specifications: a lexicon of practice, principles, and prejudices*. New York, Wokingham, England ; Reading, Mass.: ACM Press ; Addison-Wesley Pub. Co., 1995.

[Kavakli 2000] E. Kavakli, "Goal-Oriented Requirements Engineering: A Unifying Framework," *Requirements Engineering Journal*, vol. 6, pp. 237-251, 2000.

[Kruchten 2000] P. Kruchten, *The rational unified process: an introduction*, 2nd ed. Reading, Mass.; Harlow, England: Addison-Wesley, 2000.

[Larman 2001] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Second ed: Prentice Hall, 2001.

[Leffingwell 2000] D. Leffingwell and D. Widrig, *Managing software requirements: a unified approach*. Reading, Mass ; Harlow, England: Addison-Wesley, 2000.

[Letier 2002] E. Letier and A. v. Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration," Proceedings ICSE'2002 - 24th International Conference on Software Engineering, Orlando, FL, 2002.

[Letier 2002] E. Letier and A. v. Lamsweerde, "Deriving Operational Software Specifications from System Goals," FSE'10 - 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering, Charleston, NC, 2002.

[Newell 1972] A. Newell and H. A. Simon, *Human problem solving*. Englewood cliffs, N.J.: Prentice-Hall, 1972.

[Parnas 1995] D. L. Parnas and J. Madey, "Functional Documents for Computer Systems," *Science of Computing Programming*, vol. 25, pp. 41-61, 1995.

[Purao 2002] S. Purao, A. Bush, and M. Rossi, "Towards an understanding of the use of problem and design spaces during object oriented system development," *Information and Organization, Pergamon*, vol. 12, pp. 249-281, 2002.

[Regnell 1996] B. Regnell, M. Anderson, and J. Bergstrand, "A Hierarchical Use Case Model with Graphical Representation," Proceedings the IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'96), Friedrischshafen, Germany, 1996.

[Robinson 1989] W. N. Robinson, "Integrating multiple specifications using domain goals," 5th International workshop on software specification and design, 1989.

[Robinson 1994] W. N. Robinson, "Interactive Decision Support for Requirements Negotiation," *Concurrent Engineering: Research & Applications, Special Issue on Conflict Management in Concurrent Engineering, The Institute of Concurrent Engineering,*, vol. 2, pp. 237-252, 1994.

[Robinson 1990] W. N. Robinson, "Negotiation behavior during requirement specification," Proceedings of the 12th International Conference on Software Engineering, Nice, France, 1990.

[Robinson 1998] W. N. Robinson and S. Pawlowski, "Surfacing Root Requirements Interactions from Inquiry Cycle Requirements," The Third IEEE International Conference on Requirements Engineering (ICRE'98), Colorado Springs, CO, 1998.

[Ross 1977] D. T. Ross and K. E. S. Jr., "Structured analysis for requirements definition," *Transactions on Software Engineering*, vol. SE-3, pp. 6-15, 1977.

[Rubin 1992] K. S. Rubin and A. Goldberg, "Object Behavior Analysis," *Communications of the ACM*, vol. 35, pp. 48-62, 1992.

[Rumbaugh 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *ObjectOriented Modeling and Design*: Prentice Hall, 1991.

[Soloway 1984] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *Transactions on Software Engineering*, vol. SE-10, pp. 595-609, 1984.

[Sommerville 1997] I. Sommerville and P. Sawyer, *Requirements engineering: a good practice guide*. Chichester, England; New York: John Wiley & Sons, 1997.

[van Lamsweerde 2001] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," RE'01 - 5th IEEE International Symposium on Requirements Engineering, Toronto, 2001.

[van Lamsweerde 1998] A. van Lamsweerde, R. Darimont, and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering," *IEEE, Transactions on Software Engineering*, vol. 24, pp. 908-926, 1998.

[van Lamsweerde 2000] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," in *IEEE Transactions on Software Engineering*, vol. 26, 2000, pp. 978-1005.

[Weidenhaupt 1998] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenarios in System Development: Current Practice," *IEEE Software*, vol. 15, pp. 34-45, 1998.

[Yue 1987] K. Yue, "What does it mean to say that a specification is complete?," 4th International workshop on software specification and design, Monterrey,CA, 1987.

About the authors

William N. Robinson is an independent software consultant and associate professor of Computer Information Systems at Georgia State University. He can be reached at wrobinson@gsu.edu.

Greg Elofson is professor at the Graduate School of Business, University of New Orleans. He can be reached at gelofson@uno.edu.