User-Centered System Decomposition: Z-Based Requirements Clustering

Pei Hsia C. T. Hsu David C. Kung Lawrence B. Holder

Computer Science and Engineering Department
The University of Texas at Arlington
Arlington, Texas, USA

Abstract

Requirements clustering (RC) provides a different approach to system decomposition, by enabling a system to be partitioned into user-recognizable components, where each component can be used, almost independently, to satisfy part of the user's needs. Requirements clustering is essential for a software development approach called incremental delivery (ID). A successful clustering of system requirements produces a set of useful, usable, and semi-independent clusters that can be developed and delivered to the customers in increments. This paper presents a requirements clustering process based on ER modeling, scenarios, and the formal specification notation Z.

1 Introduction

Reducing the complexity of large software systems is one of the most important tasks in the entire software development process. By applying the concept of divide-and-conquer, systems can be decomposed into less complex parts, so each part can be addressed with less difficulty. Typically, a system is partitioned into smaller functional components. Functional decomposition applies the divide-and-conquer concept from a system developer's view and provides a perspective for system construction. Unfortunately, it is usually not easy to map these functional parts onto customer-recognizable components. Therefore, the customers/users are left out of the system development process because they do not understand the internal details. Furthermore, developers tend to concentrate on the functional details and easily lose sight of the objectives of the system and the goal of the product.

Incremental delivery transforms the major steps in software development, such as requirements analysis, system partitioning, and system integration, into a user-centered perspective. It partitions a whole system from its utility point of view and allows a system to be grouped into usable subsystems (called increments). Each subsystem is semi-independent from the rest of the system and can be implemented, tested, and delivered to the customer separately in a well-defined sequence according to priority, precedence relation, and other criteria. The delivered subsystems are immediately usable to the customer to fulfill part of his/her requirements. A key factor to the success of ID is to cluster requirements into increments in such a way that each increment can operate without much functional help from the other increments.

This paper presents a requirements clustering process based on ER modeling, scenarios, and the formal specification notation Z, and uses a simplified airline scheduling and reservation system to illustrate the approach. The remainder of the paper is structured as follows. Section 2 summarizes related work on requirements clustering and incremental delivery. Section 3 provides a brief explanation of some of the key concepts and terms of the Z notation. Based on this, the proposed requirements clustering process and a simplified airline scheduling and reservation system (ASR) is presented in Section 4. Section 5 summarizes the paper and our future work.

2 Related work

2.1 Incremental delivery

Traditionally, software systems have been considered a monolithic piece: no part is separable from the rest, and all software components must be present to achieve an operational system. Recently, researchers and practitioners have been advocating incremental development, and even incremental delivery, to construct and deliver software systems (e.g. [4], [7], [10], [11], [16]). Incremental delivery takes incremental development a step further by providing software systems to end-users in components. Each delivered increment supports a partial set of requirements, and its functionalities are visible to the end-users. These new approaches are conceptually appealing, because they provide both customers and management with an essential ingredient which is conspicuously absent in the monolithic development approach: progress visibility. The increments that are delivered to the customers can be examined for their functionalities and provide a foretaste of the things to come. At the same time, they serve as concrete progress achievements to the customers, management, and system developers themselves. Some additional advantages of ID are summarized in ([11], [12], [13], [15]). A somewhat similar approach to ID is called evolutionary delivery (ED) [5]. In ED, a software development project is divided into a carefully-planned sequence of releases. Early releases deliver the core functionalities, while subsequent releases add on more capabilities. ED differs from ID in that each release in ED delivers a more complete set of system capabilities than the previous releases, while in ID, the delivered increments are semi-independent of one another. Hough discussed a system development approach called rapid delivery [10], by identifying twelve issues concerning system segmentation strategies. Hepner [8] proposed an object-oriented incremental delivery approach (OOID) based on Coad's OOA and scenarios. OOID identifies similar functions by their relatedness to common objects, services, and/or message connections. Scenarios which relate to similar functions are clustered in the same group. The increment implementation order is then determined based on their interdependency.

2.2 Requirements clustering

Hsia and Yaung [12] proposed a RC algorithm based on scenario-based prototyping. This algorithm consists of two major steps. The first step generates an initial clustering from a set of scenarios, and the second step refines these clusters. The initial set of clusters is derived from the scenarios obtained by scenario-based prototyping [9] during requirements validation. The second step generates a matrix of indicator functions where the indicator functions represent the strength of relations between any requirement pair.

The success of this algorithm depends on identifying and assigning proper strengths to the relations involved between any requirement pair. The strength assignments are based solely upon the developers' understanding of the system, and are, therefore, subjective. The effort involved in the strength assignment between any two requirements is non-trivial when the number of requirements *n* become large. Without an objective and mechanical strength assignment scheme, the incurred overhead cannot be overlooked.

Hsia and Gupta [13] lessened the problem of subjectivity involved in strength assignments by using IDEF1 [19] and a methodology for building data-dominant systems called Onion [6]. Software system development using Onion normally consists of four major steps: 1) create an IDEF1 data model for the system; 2) identify abstract data types (ADTs) from this model; 3) specify, implement, and verify each ADT; and 4) implement user required functions, or external visible functions (EVFs), using the above ADTs. EVFs are classified into two categories: 1) modifying ADTs (T_m); and 2) accessing the content of ADTs (T_a). The idea is that any two EVFs in the T_m category are grouped into the same logical cluster if they change one or more common ADTs. EVFs in the T_a category do not change the state of an ADT, so each such requirement is placed in a cluster by itself. This algorithm tends to result in a larger number of clusters. Besides, the total number of clusters is fixed, and it may not be flexible enough to fit in different development contexts.

3 Definitions

We will briefly explain some key concepts and terms of the formal specification notation Z that are relevant to our following discussion. Here, we assume that the structuring of Z specifications follows the "Established Strategy" [2], namely a Z specification consists of a system state schema, an initial state schema, and a set of operation schemas.

The state schema describes all or some part of a system state. It consists of a set of *state components* and some constraining *predicates* that are defined based on those components. Each state component is associated with an implicit *value* at any given time after the system's initialization. Here at the abstract level, we need to know nothing about the internal structures of the state components and their associated values. Hence, we introduce given sets to model state components and their associated values.

$[COMPONENT, VALUE]$

Predicates are normally built upon state components. A predicate may refer to zero or more state components. The same state component may appear in one predicate more than once. However, for our purpose, we do not need to distinguish them. Therefore, a predicate can be modeled as

$| \; Predicate \mathrel{\widehat{=}} \mathbb{P}\, COMPONENT$

The state of a system at any given time is determined by the states of all its components at that time.

Components' states are modeled as a function from *COMPONENT* to *VALUE*.

$$\begin{array}{|l} \mathit{ComponentState} \mathrel{\widehat=} \mathit{COMPONENT} \rightarrow \mathit{VALUE} \end{array}$$

A system's state is thus a combination of its components' states.

$$\begin{array}{|l} \mathit{SystemState} \mathrel{\widehat=} \mathbb{P}\, \mathit{ComponentState} \end{array}$$

Operation schemas, sometimes called state transition schemas, are used to define all the operations that are applied to the system state. An operation schema consists of a pre-condition and a post-condition. Both the pre- and post-conditions contain a set of logically conjuncted and/or disjuncted predicates. Operation schemas and their pre and post-conditions are modeled as

$$\begin{array}{|l} \mathit{PreCond}, \mathit{PostCond} : \mathbb{P}\, \mathit{Predicate} \\ \\ \mathit{Oper} \mathrel{\widehat=} \mathit{PreCond} \mathrel{+\!\!\!\!\rightarrow} \mathit{PostCond} \end{array}$$

We categorize operation schemas into three main types: 1) delta schemas, 2) xi schemas, and 3) utility schemas. A delta schema represents a successful execution of an operation. The result of successfully executing a delta schema will change the values of one or more state components, namely components' state changes. Schema *MakeReservation* is an example of a delta schema. A xi schema shows no changes of any one of the state components. It normally specifies query results and describes situations that involves a reference to any of the state components. Schema *SeatAvailable* is an example of a xi schema. The third category of operation schemas–utility schemas–are schemas that do not refer to any one of the state components.

Delta and xi schemas relate to system state changes. The difference between these two types of schemas is that in delta schemas, the states of at least one of the state components are changed, while none of the state components change their states in xi schemas. These two operation schemas are modeled as total functions from *SystemState* to *SystemState*.

$$\begin{array}{|l} \mathit{delta}, \mathit{xi} : \mathit{SystemState} \rightarrow \mathit{SystemState} \\ \hline \forall\, s : \mathit{SystemState} \bullet \mathit{delta}\ s \neq s \\ \qquad\qquad\qquad\quad \mathit{xi}\ s = s \end{array}$$

A system specification typically includes these three types of operation schemas. Delta schemas specify the functional and behavioral aspects of a system's requirements, while xi and utility schemas complement delta schemas in specifying exceptional cases when the execution of a delta schema fails. The operation schemas that describe the successful cases and various exceptional cases are normally combined using schema conjunction and schema disjunction to produce a complete specification.

As previously mentioned, predicates refers to zero or more state components to specify constraints, pre-conditions, and/or post-conditions. We identify two different types of state component references. First, a predicate may use the current implicit values of the state components. In this case, we call it a *read* reference, since no change of these components' values are made. Second, a predicate may refer to one or more state components and change the values of these components. Here, the reference to any one of the state components is called a *write* reference. Both read and write references refer to at least one state components. Read and write references are modeled as partial functions from *Oper* to $\mathbb{P}_1$ *COMPONENT*. Functions *write* and *read* return the set of state components that are write-referenced and read-referenced by an operation schema, respectively.

$$\begin{array}{|l} \mathit{write} : \mathit{Oper} \mathrel{+\!\!\!\!\rightarrow} \mathbb{P}_1\, \mathit{COMPONENT} \\ \hline \forall\, s, s' : \mathit{SystemState};\ op : \mathit{Oper} \mid s' = \mathit{delta}\ s \bullet \\ \mathit{write}\ op = \{c : \mathit{COMPONENT};\ v, v' : \mathit{VALUE} \mid \\ \qquad (c \mapsto v) \in s \ \wedge\ (c \mapsto v') \in s' \ \wedge\ v \neq v' \bullet c\} \end{array}$$

$$\begin{array}{|l} \mathit{read} : \mathit{Oper} \mathrel{+\!\!\!\!\rightarrow} \mathbb{P}_1\, \mathit{COMPONENT} \\ \hline \forall\, s, s' : \mathit{SystemState};\ op : \mathit{Oper};\ \mid s' = (\mathit{delta}\ s\ \cup\ \mathit{xi}\ s) \bullet \\ \mathit{read}\ op = \{c : \mathit{COMPONENT};\ p : \mathit{Predicate} \mid c \in p \\ \qquad \wedge\ (p \in \operatorname{dom} op \vee (p \in \operatorname{ran} op \wedge c \notin \mathit{write}\ op)) \bullet c\} \end{array}$$

The interrelatedness between a state component and a schema is defined based on the number of read and write references.

$$\begin{array}{|l} \mathit{comp\_op} : \mathit{COMPONENT} \times \mathit{Oper} \rightarrow \mathbb{N} \\ \hline \forall\, op : \mathit{Oper};\ c : \mathit{COMPONENT};\ p : \mathit{Predicate} \mid c \in p \\ \qquad\qquad \wedge\ (p \in \operatorname{dom} op\ \vee\ p \in \operatorname{ran} op) \bullet \\ \mathit{comp\_op}(c, op) = 2 * \#(\mathit{write}\ op) + \#(\mathit{read}\ op \setminus \mathit{write}\ op) \end{array}$$

4 The requirements clustering process

The requirements clustering process consists of three major activities. First, a set of scenarios is prepared in an attempt to cover all of the possible ways that a system can be used. Each scenario is represented as a scenario tree. A data model of the system is then developed and represented as an ER diagram. Second, the ER model is mapped into a Z state schema. A set of related operation schemas is then identified and specified for each scenario. Third, a six-step requirements clustering algorithm is used to further group together strongly coupled operation schemas into clusters. A schematic representation of the requirements clustering process is shown in Figure 1.

In this paper, we focus on activities two and three, namely, how the ER model and scenarios can help

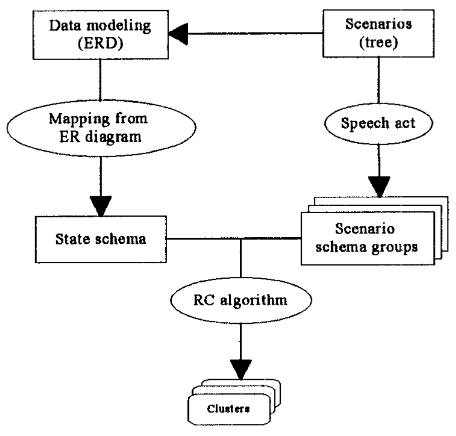

Figure 1: The requirements clustering process

construct a Z specification and how related operation schemas can be clustered into coherent increments. A simplified airline scheduling and reservation system is used to further illustrate the approach.

4.1 Constructing the system state schema

In general, a Z specification starts by defining the system state schema along with the given sets and the global constants. ER modeling provides a simple way to begin constructing the state schema. It helps the specifiers better understand the problem before jumping into the details of the state schema specification. A well-defined data model provides a better chance for constructing a good state schema.

Semmens et al. [22] propose a mapping scheme that allows ER diagrams to be mapped into Z state schemas. For our purpose, we use a slightly different mapping scheme. First, types are declared for each attribute. These types are then specified as given sets. An *entity schema* is introduced to include all the non-key attributes of an entity. Each entity is then specified as either a partial or a total function from its key attribute to the entity schema. For example, as shown in Figure 3, entity 'Flight' is specified as an entity schema and a partial function (injection) from its key attribute to the entity schema.

Flight

$$\begin{array}{l} airline : AIRLINE \\ origin, dest : AIRPORT \\ date : DATE \\ depTime, arrTime : TIME \\ craft : CRAFT \end{array}$$

$$flight : FID \rightarrowtail\!\!\!\!\!\!+\; Flight$$

Entities with a single (key) attribute can simply be modeled as given sets.

A binary relationship can be specified as either a function or a relation from the key attribute of an entity to the key attribute or the entity schema of another entity. One-to-one relationships can be specified as injections. For example, "a reservation takes a seat" is modeled as an injection from RESID to Seat.

$$take : RESID \rightarrow Seat$$

One-to-many or many-to-one relationships can be specified as either partial or total functions. As shown in Figure 3, the many-to-one relationship 'reserved_by' between 'Reservation' and 'Passenger' can be specified in one of the following two ways:

$$\begin{array}{l} reservedby : RESID \rightarrow Passenger \\ reserve : SSN \nrightarrow \mathbb{P}\, RESID \end{array}$$

The choice between these two relies on the specifier's decision on which one leads to a simple and better ensuing operation specification.

Figure 2 shows the ER diagram of the ASR system. The state schema is constructed by mapping from the ER diagram. The specification of the state schema is included in the appendix.

4.2 Scenario-based specification

A system seldom exists on its own. It is either a part of a larger system under which the system cooperates with other systems to achieve a broader goal, or it is an autonomous working unit that can be used by its users independently. The external entities, either other peer systems or the users, that a system interacts with are called *agents*.

Types of interactions
The interaction between a system and its agents can be thought of as a restricted form of human conversation. The basic unit of interaction is a *speech act* [1]. A speech act, in general, has two components: a propositional content and an illocutionary point (force). The propositional content specifies what is being requested, warned about, ordered, etc. Each type of speech act has a point or a purpose essential to its being a speech act of that type.

Based on the illocutionary point, Searle ([20], [21]) classified speech acts into five categories: assertives, directives, commissives, expressives, and declaratives. The purpose of an *assertive* speech act is for the speaker to "tell how things are" to the hearer. A directive is a speech act where the speaker gets the hearer to do something. Commissive speech acts are to commit the speaker to do something. The expressive speech acts are to express the speaker's feelings and attitudes. A declarative is to bring about changes in the world by saying so.

Constructing operation schemas using scenarios
Recently, scenarios has been actively used in many areas, especially in the areas of human computer interaction and software engineering (e.g., [3], [17], [18]). So far, there is no clear consensus about the definition of scenarios. Here, we define scenarios as sequences of interactions in terms of speech acts between a system and its agents. The definition of scenarios in terms of the theory of speech act suggests a systematic procedure for constructing operation schemas. The procedure is divided into three steps. First, each scenario is represented as a scenario tree [14]. Each node in the scenario tree represents a state, while an edge indicates an occurrence of an event. Here, we consider only the events (or system-agent interactions) and do not consider the details of the states. The labels of all the nodes are, therefore, suppressed. Second, all the transition centers (TCs) within each scenario must be identified. A *transition center* is a node that represents the focus of the interaction between a system and its agents. It indicates system's (re) actions and will possibly trigger the transition of its states to fulfill agent's requests. To locate transition centers, look for

- commissives from system to agents, and
- nodes with outdegree greater than 1, or at least one of its outgoing paths result in a system state change.

Third, the scope of each operation schema must be determined. A transition center is the center for a set of potentially related operation schemas. To determine the scope (or the granularity) of each operation schema, we start from the transition center and look for the input boundary and the output boundary by traversing upward and downward, respectively, along a path in the scenario tree. Each path between an input boundary and an output boundary is a potential operation schema.

Input and output boundaries are open to interpretation. Different specifiers may select different points in the path as boundary locations. When looking for the input boundary, we suggest commissives from agents to system or directives from system to agents while commissives or assertives from system to agents are potential output boundary locations. As shown in Figure 2, two TCs, labeled as TC_1 and TC_2, are identified. For TC_1, one input boundary (I_1) and two output boundaries (O_{11} and O_{12}) are chosen. Likewise, the input and output boundaries for TC_2 are chosen as I_2 and O_{21} and O_{22}, respectively. There are four different paths: 1) I_1 to O_{11}; 2) I_1 to O_{12}; 3) I_2 to O_{21}; and 4) I_2 to O_{22}. These four paths suggest four potential operation schemas: 1) *SeatAvailable*, 2) *NoSeatAvailable*, 3) *MakeReservation*, and 4) *ReservationExist*. The operation schemas that are derived from a scenario tree are referred to as a ***scenario schema group***. Scenario schema groups are further classified into two types: delta schema group and xi schema group. A delta schema group includes at least one delta schema, while a xi schema group does not. The eleven scenarios and their associated scenario schema groups of the ASR system are listed in the appendix.

4.3 The requirements clustering algorithm

The final activity of the requirements clustering process is a six-step RC algorithm:

Step 1: Construct the component-delta (CD) matrix.
A CD matrix is constructed by assigning an integer $CD_{ij} = comp\_op(i,j)$ to entry (i,j) where state component i is referenced by delta schema group j. CD_{ij} represents a syntactical measure of the interdependency (CD coupling) between a state component and a delta schema group.

The CD matrix for the ASR system is shown in Figure 4(a). To explain how this matrix is constructed, let us examine the $(passenger, \text{S6})$ entry of the matrix. One read reference (in *ReservationExist*) and one write reference (in *MakeReservation*) of the state component *passenger* are identified in delta schema group S6. Therefore, the $\text{CD}_{passenger,S6}$ entry of the matrix is assigned to be 3 to indicate the coupling strength between delta schema group S6 and the state component *passenger*.

Step 2: Construct the delta-delta (DD) coupling indicator matrix.
The DD coupling indicator matrix is constructed based on the CD matrix. The (i,j) entry of the DD matrix is assigned an integer DD_{ij} derived from the CD matrix. That is

$$DD_{ij} = \sum_{k=1}^{n} \text{Min}(CD_{ki}, CD_{kj})$$

where n = number of state components. The value of DD_{ij} indicates the relative strength of interdependency between delta schema groups i and j.

The DD coupling indicator matrix for the ASR system is shown in Figure 4(b). The value of the DD_{ij} entry is determined by looking at columns i and j of the CD matrix. It is derived by summing up the minimum positive values of CD_{ki} and CD_{kj}, for k = 1 to the number of state components (11 in this example). For example, $DD_{S1,S2} = min(4,5) + min(3,2) + min(2,2) + min(3,2) + min(2,2) = 12$

Step 3: Build the DD coupling spectrum.
By scanning through the DD matrix for different values of DD_{ij}'s, a sequence of integers ranging from the highest DD_{ij} to the lowest one is identified and are referred to as the *DD coupling spectrum.* A higher value of DD_{ij} represents a higher degree of coupling between delta schema group i and j. The DD coupling spectrum provides a road map for selecting the desired number of clusters. Each element in the DD spectrum is referred to as a *DD coupling strength level* ξ. The DD coupling spectrum for the ASR system is <14,12,8,7,5,3,2>.

Step 4: Cluster delta schema groups.
For a chosen ξ, cluster delta schema groups i and j with the value of DD_{ij} greater than or equal to ξ. The result is a set of clusters in which the delta schema groups within each cluster have DD coupling strength level of at least ξ. Any value in the DD coupling spectrum can be chosen as a reference DD coupling strength level ξ for clustering related delta schema groups. By choosing a smaller value of ξ, delta schema groups tend to be clustered together. This will lead to a smaller number of clusters because each cluster will cover more delta schema groups. In the ASR system, let us assume that the DD coupling strength level ξ is set to 12. The RC algorithm will produce three clusters:

- $ASR_1 = \{S1, S2\}$
- $ASR_2 = \{S3\}$
- $ASR_3 = \{S6, S8\}$

Step 5: Cluster the remaining xi schema groups.
For the rest of the xi schema groups, two options are possible:

1. Each xi schema group can be an independent cluster of its own.
2. Assign a xi schema group G to a cluster L, which includes a delta schema group D_j that has the largest value of GD_j. The value of GD_j, indicating the coupling strength between xi schema group G and cluster L, is determined by

   $$GD_j = \sum_{i \in M_G} CD_{ij}$$

   where M_G is the set of state components that are referenced by G. In case of a tie, compare the second-highest GD_j.

The choice between Options (a) and (b) is somewhat application-dependent. However, to better control the number of clusters, we suggest Option (b) and make the adjustment of the final number of clusters during the last step.

The remaining xi schema groups, and the state components they reference for the ASR system are listed in Figure 4(c). Here, we choose Option (b), namely, assign a xi schema group G to a cluster L, which has the largest value of GD_j. The remaining xi schema groups and the clusters to which they are assigned are listed in Figure 4(c). The result of the requirements clustering algorithm before the final adjustment is

- $ASR_1 = \{S1, S2, S7, S9, S10, S11\}$
- $ASR_2 = \{S3\}$
- $ASR_3 = \{S4, S5, S6, S8\}$

ASR_1 represents a subsystem that deals with the details of flights. It supports both the airline company and the travel agent's operations. The airline company can schedule a new flight and cancel a scheduled flight. A travel agent may find a flight, request a seat price, and request the arrival and departure times of a flight. ASR_2 includes a single delta schema group S3. It allows the airline company to change the air fare. ASR_3 mainly focuses on the operations that are related to seat reservations. It supports the operations of both the airline company and the travel agents. A travel agent can make or cancel a reservation, while the airline company is interested in requesting how many reservations have been made and who made the reservations in a flight.

Step 6: Consider making a final adjustment.
Steps 1 to 5 are mechanical, namely without any involvement of human judgment and decision except deciding on the value of ξ. The final number of clusters derived from the previous five steps need not perfectly reflect customer's and/or developer's needs. Adjustment of the total number of clusters may be conducted by considering such factors as "a pre-determined number of clusters based on customers' needs," "a shortest time-to-market delivery strategy," "a cost-effective number of clusters for minimizing future maintenance effort and cost [15].

In considering making the final adjustment, ASR_1 can be further divided into two clusters:

- ASR_4: $\{S1, S2\}$
- ASR_5: $\{S7, S9, S10, S11\}$

ASR_4 represents a subsystem that are related to the operations of the airline company. ASR_5 somehow relates to travel agents' operations. These two clusters, interested in by different user groups, are coupled by referencing similar sets of state components. Strong coupling indicates a possible increase of the extra code [15] in making these two clusters semi-independent of each other. This is the point that the developers and the customers have to decide in making the final adjustment, by considering customer's needs and the future software maintenance.

ASR_2 includes a single delta schema group S3 (i.e., change fare). It becomes an independent cluster, because it has a loose coupling with other delta schema groups. As we can expect, it could be a small cluster compared with the size of other clusters. To save future maintenance efforts, adjustment may be needed to produce a final set of clusters that has a uniform cluster size distribution [15]. S3 can be included in ASR_4, because only the airline company can change the fare. After making the final adjustment, the RC algorithm produces three clusters:

- ASR_3: {S4, S5, S6, S8}
- ASR_4: {S1, S2, S3}
- ASR_5: {S7, S9, S10, S11}

As illustrated in the results of the example, the final number of clusters and the content of each cluster can be decided mechanically by Steps 1 to 5 of the RC algorithm. However, they can be adjusted to meet specific needs. Basically, the DD coupling spectrum provides a road map for choosing from the desired number of clusters. Any DD coupling strength level can be chosen to produce different number and contents of clusters. This process is mechanical and coarse-grained in nature. The final number of clusters and the contents of each cluster may be fine-tuned by considering factors of interest to fit specific purposes.

5 Concluding Remarks

We have presented a requirements clustering process based on ER modeling, scenarios, and the formal specification notation Z. A simplified airline scheduling and reservation system has been used as an example to illustrate the proposed approach. The RC process provides a flexible scheme to cluster related requirements so that a final optimal number of clusters can be achieved. The proposed approach has the following advantages:

- By incorporating formal specification notations, using Z as our first attempt, into requirements clustering, it eliminates the problem of subjectivity and minimizes the strength assignment overhead when the number of requirements become large. The use of formal notation facilitates the automation of the six-step RC algorithm.
- By providing system decomposition from the user's perspectives, it leads to user-recognizable clusters.
- It allows the final number of clusters to be adjusted, by choosing an appropriate DD coupling strength level from the strength spectrum to fulfill customers' need, to adapt to a specific development environment, or to minimize future maintenance cost.
- The use of scenarios in constructing operation schemas provides a way to connect and show the temporal relationship between related operation schemas. The role played by each operation schema in a typical use of the system can be easily visualized. Furthermore, scenarios narrow the gap between developers and customers. Developers and customers can validate system requirements by tracing through each scenario and the associated scenario schema group. Each scenario indeed is the basic unit of meaningful communication between the developers and the customers.

The steps of the RC algorithm are well defined, so that it is possible to automate the entire RC process. We are currently developing a CASE tool to support incremental delivery that includes four major subsystems: 1) a requirements tool; 2) a scenarios tool; 3) a rapid prototyping tool; and 4) an incremental delivery planing and maintenance tool. Implementation of the RC algorithm is an essential part of the requirements tool.

Acknowledgments

We gratefully acknowledge Dr. Ali Mili for his generous help in verifying the correctness of some of the Z specification. We would also like to thank Mr. Liang Li for reviewing Figures 2 and 3 and all the anonymous referees for their insightful comments on an earlier draft of this paper which were indispensable in obtaining this final result.

References

[1] Austin, J.L., "How To Do Things With Words," 2nd ed., Urmson, J.O. and Sbisa, M. ed., Cambridge, MA: Harvard University Press, 1962.

[2] Barden, R., Stepney, S., and Cooper, D., "Z in Practice," Prentice-Hall, 1994.

[3] Carroll, J.M. (ed.), " Scenario-Based Design: Envisioning Work and Technology in System Development," John Wiley & Sons, Inc., 1995.

[4] Gilb, T., "Evolutionary Delivery versus the Waterfall Model," Software Engineering Notes, vol. 10, no. 3, 1985, pp. 49-62.

[5] Gilb, Tom., "Principles of Software Engineering Management," Addison-Wesley, 1988.

[6] Gupta, A.P., "Onion: A Development Methodology for Data-Dominant Systems," Ph.D. dissertation, The University of Texas at Arlington, March 1990.

[7] Hekmatpour, S., "Experience with Evolutionary Prototyping in a Large Software Project," Software Engineering Notes, vol. 12, no. 1, 1987, pp. 38-41.

[8] Hepner, M.M.,"An Object-oriented Approach to Incremental Delivery of Software Systems," Masters thesis, The University of Texas at Arlington, 1991.

[9] Hooper, J.W. and Hsia, p., "Scenario-Based Prototyping for Requirements Identification," ACM SIGSOFT, Software Engineering Notes, vol. 7, no. 5, Dec. 1982, pp. 88-93.

[10] Hough, D., "Rapid Delivery: An Evolutionary Approach for Application Development," IBM Systems Journal, vol. 32, no. 3, 1993, pp. 397-419.

[11] Hsia, P., Yaung, A.T., and Jiam, S.H., "Requirements Clustering for Incremental Construction of Software Systems," Proc. COMPSAC '86, Oct. 1986.

[12] Hsia, P. and Yaung, A.T., "Another Approach to System Decomposition: Requirements Clustering," Proc. COMPSAC '88, October, 1988.

[13] Hsia, P. and Gupta, A., "Incremental Delivery Using Abstract Data Types and Requirements Clustering," ICSI '92.

[14] Hsia, P. et al., "Formal Approach to Scenario Analysis," IEEE Software, March 1994, pp. 33-41.

[15] Hsia, P., Hsu, C.T., Kung, D.C., and Yaung, A.T., "The Impact of Incremental Delivery on Maintenance Effort: An Analytical Study," Proc. of ESEC '95, September 26-28, Spain, 1995, pp. 405-422.

[16] Ichikawa, H., Itoh, M., and Kato, J., "SDE: Incremental Specification and Development of Communications Software," IEEE Transactions on Computers, April 1991, pp. 553-561.

[17] Jacobson, I., "Object-Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley, 1992.

[18] Rumbaugh, J. et al., "Object-Oriented Modeling and Design," Prentice Hall, 1991.

[19] Ruoff, K.L., "Practical Application of the IDEF1 as a Database Development Tool," Proc. Intl. Conf. on Data Engineering, 1984.

[20] Searle, J.R., "Speech Acts: An Essay in the Philosophy of Language," Cambridge University Press, 1969.

[21] Searle, J.R., "A Taxonomy of Illocutionary Acts," In Searle, J.R. (ed), Expression and Meaning: Studies in the Theory of Speech Acts, Cambridge University Press, 1979, pp. 1-29.

[22] Semmens, L. and Allen, P., "Using Yourdon and Z: an Approach to Formal Specification," Proc. Z User Workshop, Oxford, 1990, pp. 228-253.

Appendix

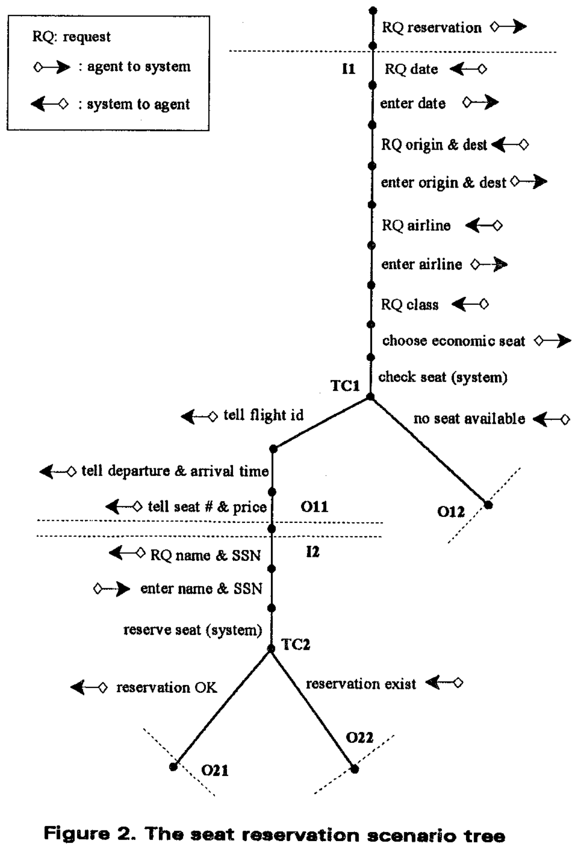

Figure 2. The seat reservation scenario tree

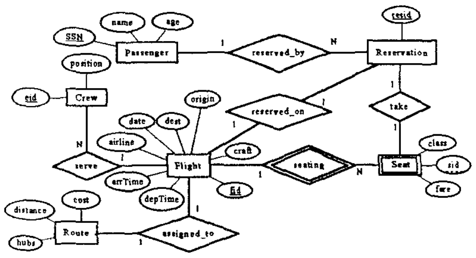

Figure 3. The ER model of the ASR system

	S1	S2	S3	S6	S8
flight	4	5	2	4	3
crew	3	2			
route	2				
assignedto	2	2			
passenger				3	5
seat	1			3	
serve	3	2			
reservedby				3	4
reservedon		2		3	4
take				4	2
seating	2	2	1	2	

(a) The CD matrix

	S1	S2	S3	S6	S8
S1	x	12	3	7	3
S2		x	3	8	5
S3			x	3	2
S6				x	14
S8					x

(b) The DD coupling indicator matrix

xi schema group	state components	max GD_i	assigned to
S4	*flight, reservedon, reservedby*	11	ASR_3
S5	*flight, reservedon*	7 (tie), 7	ASR_3
S7	*flight*	5	ASR_1
S9	*flight, seating*	7	ASR_1
S10	*flight*	5	ASR_1
S11	*flight*	5	ASR_1

(c) Clustering the remaining xi schema groups

Figure 4. Requirements clustering for the ASR system

- Scenarios:

  S1: Schedule a new flight.
  S2: Cancel an existent flight.
  S3: Change air fare.
  S4: Request the passenger list in a flight.
  S5: Request total number of reservations in a flight.
  S6: Make a reservation.
  S7: Find a flight.
  S8: Cancel a reservation.
  S9: Request seat price.
  S10: Request the arrival time of a flight.
  S11: Request the departure time of a flight.

1. The state schema

- Entities are mapped to schema types or given sets.

```
┌ Flight ──────────────────────────
│ airline : AIRLINE
│ origin, dest : AIRPORT
│ date : DATE
│ depTime, arrTime : TIME
│ craft : CRAFT
└──────────────────────────────────
```

```
┌ Crew ────────────────────────────
│ position : POSITION
└──────────────────────────────────
```

```
┌ Route ───────────────────────────
│ hubs : ℙ AIRPORT
│ distance : DISTANCE
│ cost : COST
├──────────────
│ #hubs = 2
└──────────────────────────────────
```

```
┌ Passenger ───────────────────────
│ name : NAME
│ age : AGE
└──────────────────────────────────
```

```
┌ Seat ────────────────────────────
│ class : CLASS
│ fare : ℕ
└──────────────────────────────────
```

Entity "Reservation" is modeled as given set *RESID*.

```
┌ Entity ──────────────────────────
│ flight : FID ⤔ Flight
│ crew : EID ⇸ Crew
│ route : RID ⤔ Route
│ seat : SID ⇸ Seat
│ passenger : SSN ⇸ Passenger
└──────────────────────────────────
```

- Relationships are mapped to functions or relations.

```
┌ Relationship ────────────────────
│ reservedby : RESID → Passenger
│ reservedon : RESID → Flight
│ take : RESID → Seat
│ seating : FID → bag Seat
│ assignedto : RID ⤔ Flight
│ serve : EID ⇸ FID
├──────────────────────────────────
│ dom reservedon = dom reservedby = dom take
└──────────────────────────────────
```

- The state schema and initial state schema of the ASR system are

```
┌ ASR ─────────────────────────────
│ Entity
│ Relationship
└──────────────────────────────────
```

```
┌ ASRInit ─────────────────────────
│ ASR′
├──────────────
│ flight′ = crew′ = route′ = seat′ = passenger′ = ∅
│ reservedby′ = reservedon′ = take′ = seating′ =
│                 assignedto′ = serve′ = ∅
└──────────────────────────────────
```

2. Scenario schema groups and operation schemas

S1: *ScheduleFlight, AssignCrew, AssignRoute, UnknownRoute, BusyCrew, FlightExist*
S2: *FlightExist, CancelFlight, HasReservation, NoSuchFlight*
S3: *ChangeFare, NoSuchFlight*
S4: *NoSuchFlight, FlightExist, PassengerList*
S5: *NoSuchFlight, FlightExist, TotalReservation*
S6: *SeatAvailable, MakeReservation, ReservationExist, NoSeatAvailable*
S7: *FindFlight*
S8: *NoSuchPassenger, KnownPassenger, NoReservation, KnownReservation, CancelReservation*
S9: *SeatPrice*
S10: *NoSuchFlight, FlightExist, ArrivalTime*
S11: *NoSuchFlight, FlightExist, DepartureTime*

```
┌ ScheduleFlight ──────────────────
│ ΔASR
│ a? : AIRLINE
│ f? : FID
│ orig?, des? : AIRPORT
│ d? : DATE
│ dep?, arr? : TIME
│ cr? : CRAFT
│ st? : bag Seat
├──────────────
│ f? ∉ dom flight
│ st? ⊑ ran seat
│ let ft ≙ (μ Flight | airline = a? ∧ origin = orig?
│          ∧ dest = des? ∧ date = d? ∧ depTime = dep?
│          ∧ arrTime = arr? ∧ craft = cr?) •
│    flight′ = flight ∪ {f? ↦ ft}
│    seating′ = seating ∪ {f? ↦ st?}
└──────────────────────────────────
```

FlightExist $\widehat{=}$ [ΞASR; $f? : FID \mid f? \in \mathrm{dom}\ flight$]

```
┌ AssignCrew ──────────────────────
│ ΔASR
│ e? : EID
│ f? : FID
├──────────────
│ f? ∈ dom flight
│ e? ∈ dom crew
│ e? ∉ dom serve
│ serve′ = serve ∪ {e? ↦ f?}
└──────────────────────────────────
```

```
┌ BusyCrew ────────────────────────
│ ΞASR
│ e? : EID
│ rep! : REPORT
├──────────────
│ e? ∈ dom crew
│ e? ∈ dom serve
│ rep! = 'Crew already in service'
└──────────────────────────────────
```

```
┌ AssignRoute ─────────────────────
│ ΔASR
│ f? : FID
│ r? : RID
├──────────────
│ f? ∈ dom flight
│ r? ∈ dom route
│ r? ∉ dom assignedto
│ assignedto′ = assignedto ∪ {r? ↦ flight f?}
└──────────────────────────────────
```

```
┌ UnknownRoute ────────────────────
│ ΞASR
│ r? : RID
│ rep! : REPORT
├──────────────
│ r? ∉ dom route
│ rep! = 'Unknown route'
└──────────────────────────────────
```

```
┌ CancelFlight ────────────────────
│ ΔASR
│ f? : FID
├──────────────
│ f? ∈ dom flight
│ reservedon⁻¹ ⦇ flight f? ⦈ = ∅
│ flight′ = {f?} ⩤ flight
│ seating′ = {f?} ⩤ seating
│ assignedto′ = assignedto ⩥ (flight f?)
│ serve′ = serve ⩥ f?
└──────────────────────────────────
```

```
┌ HasReservation ──────────────────
│ ΞASR
│ f? : FID
│ rep! : REPORT
├──────────────
│ reservedon⁻¹ ⦇ flight f? ⦈ ≠ ∅
│ rep! = 'Flight has reservations'
└──────────────────────────────────
```

```
┌ NoSuchFlight ────────────────────
│ ΞASR
│ f? : FID
│ rep! : REPORT
├──────────────
│ f? ∉ dom flight
│ rep! = 'No such flight'
└──────────────────────────────────
```

ChangeFare

$$\begin{array}{|l}
\Delta ASR \\
f? : FID \\
c? : CLASS \\
fa? : FARE \\
\hline
f? \in \operatorname{dom} flight \\
f? \in \operatorname{dom} seating \\
\forall s : Seat \mid s \text{ in } seating\, f? \wedge s.class = c? \bullet s.fare = fa? \\
\hline
\end{array}$$

PassengerList

$$\begin{array}{|l}
\Xi ASR \\
f? : FID \\
p! : \mathbb{P}\, Passenger \\
\hline
f? \in \operatorname{dom} flight \\
p! = \{\forall r : RESID \mid flight^{-1} \llparenthesis\, reservedon\; r \,\rrparenthesis = f? \bullet \\
\qquad\qquad reservedby\; r\} \\
\hline
\end{array}$$

TotalReservation

$$\begin{array}{|l}
\Xi ASR \\
f? : FID \\
out! : \mathbb{N} \\
\hline
f? \in \operatorname{dom} flight \\
out! = \# \{\forall f : Flight \mid flight^{-1} \llparenthesis\, f \,\rrparenthesis = f? \bullet \\
\qquad\qquad\qquad reservedon^{-1} \llparenthesis\, f \,\rrparenthesis\} \\
\hline
\end{array}$$

SeatAvailable

$$\begin{array}{|l}
\Xi ASR \\
d? : DATE \\
orig?, des? : AIRPORT \\
a? : AIRLINE \\
c? : CLASS \\
fid! : FID \\
dep!, arr! : TIME \\
seatid! : SID \\
fa! : FARE \\
\hline
\exists f : Flight;\; s : Seat \mid f \in \operatorname{ran} flight \wedge \\
\qquad s \text{ in } seating(flight^{-1} \llparenthesis\, f \,\rrparenthesis) \\
\qquad \wedge f.date = d? \wedge f.origin = orig? \wedge f.dest = des? \\
\qquad \wedge f.airline = a? \wedge s.class = c? \wedge s \notin \operatorname{ran} take \bullet \\
\quad fid! = flight^{-1} \llparenthesis\, f \,\rrparenthesis \\
\quad dep! = f.depTime \\
\quad arr! = f.arrTime \\
\quad seatid! = seat^{-1} \llparenthesis\, s \,\rrparenthesis \\
\hline
\end{array}$$

MakeReservation

$$\begin{array}{|l}
\Delta ASR \\
fid? : FID \\
c? : CLASS \\
fa? : FARE \\
s? : SSN \\
n? : NAME \\
ag? : \mathbb{N} \\
r! : RESID \\
sid! : SID \\
\hline
r! \notin \operatorname{dom} reservedby \\
\text{let } st \mathrel{\widehat=} (\mu\, Seat \mid class = c? \wedge fare = fa?) \wedge \\
\qquad pr = (\mu\, Passenger \mid name = n? \wedge age = ag?) \bullet \\
\quad passenger' = passenger \cup \{s? \mapsto pr\} \\
\quad reservedby' = reservedby \cup \{r! \mapsto pr\} \\
\quad take' = take \cup \{r! \mapsto st\} \\
\quad reservedon' = reservedon \cup \{r! \mapsto flight\, f?\} \\
\quad seat' = seat \cup \{sid! \mapsto st\} \\
\hline
\end{array}$$

NoSeatAvailable

$$\begin{array}{|l}
\Xi ASR \\
d? : DATE \\
orig?, des? : AIRPORT \\
a? : AIRLINE \\
c? : CLASS \\
rep! : REPORT \\
\hline
\neg\, (\exists f : Flight;\; s : Seat \mid f \in \operatorname{ran} flight \\
\qquad \wedge s \text{ in } seating(flight^{-1} \llparenthesis\, f \,\rrparenthesis) \wedge f.date = d? \\
\qquad \wedge f.origin = orig? \wedge f.dest = des? \wedge f.airline = a? \\
\qquad \wedge s.class = c? \wedge s \notin \operatorname{ran} take) \\
rep! = \text{‘No seat available’} \\
\hline
\end{array}$$

ReservationExist

$$\begin{array}{|l}
\Xi ASR \\
s? : SSN \\
f? : FID \\
rep! : REPORT \\
\hline
\exists r : RESID \mid r = reservedby^{-1} \llparenthesis\, passenger\; s? \,\rrparenthesis \wedge \\
\qquad\qquad r = reservedon^{-1} \llparenthesis\, flight\, f? \,\rrparenthesis \\
rep! = \text{‘Reservation exist’} \\
\hline
\end{array}$$

FindFlight

$$\begin{array}{|l}
\Xi ASR \\
d? : DATE \\
orig?, des? : AIRPORT \\
dep! : \mathbb{P}\,(FID \times TIME) \\
\hline
dep! = \{\forall f : Flight \mid f \in \operatorname{ran} flight \wedge f.date = d? \\
\qquad\quad \wedge f.origin = orig? \wedge f.dest = des? \bullet \\
\qquad\qquad\qquad flight^{-1} \llparenthesis\, f \,\rrparenthesis \mapsto f.depTime\} \\
\hline
\end{array}$$

NoSuchPassenger

$$\begin{array}{|l}
\Xi ASR \\
s? : SSN \\
rep! : REPORT \\
\hline
s? \notin \operatorname{dom} passenger \\
rep! = \text{‘No such passenger’} \\
\hline
\end{array}$$

$$KnownPassenger \mathrel{\widehat=} [\Xi ASR;\; s? : SSN \mid s? \in \operatorname{dom} passenger]$$

KnownReservation

$$\begin{array}{|l}
\Xi ASR \\
f? : FID \\
s? : SSN \\
\hline
reservedon^{-1} \llparenthesis\, flight\, f? \,\rrparenthesis \in reservedby^{-1} \llparenthesis\, passenger\; s? \,\rrparenthesis \\
\hline
\end{array}$$

NoReservation

$$\begin{array}{|l}
\Xi ASR \\
f? : FID \\
s? : SSN \\
rep! : REPORT \\
\hline
reservedon^{-1} \llparenthesis\, flight\, f? \,\rrparenthesis \notin reservedby^{-1} \llparenthesis\, passenger\; s? \,\rrparenthesis \\
rep! = \text{‘No reservation for such passenger’} \\
\hline
\end{array}$$

CancelReservation

$$\begin{array}{|l}
\Delta ASR \\
s? : SSN \\
f? : FID \\
\hline
\text{let } r \;=\; reservedon^{-1} \llparenthesis\, flight\, f? \,\rrparenthesis = \\
\qquad reservedby^{-1} \llparenthesis\, passenger\; s? \,\rrparenthesis \\
reservedby' = \{r\} \ndres reservedby \\
reservedon' = \{r\} \ndres reservedon \\
take' = \{r\} \ndres take \\
\hline
\end{array}$$

SeatPrice

$$\begin{array}{|l}
\Xi ASR \\
a? : AIRLINE \\
c? : CLASS \\
fa! : FARE \\
\hline
\forall f : FID;\; st : Seat \mid f \in \operatorname{dom} flight \wedge st \text{ in } seating\, f \\
\qquad\quad \wedge (flight\, f).airline = a? \wedge st.class = c? \bullet \\
\qquad\qquad\qquad fa! = st.fare \\
\hline
\end{array}$$

ArrivalTime

$$\begin{array}{|l}
\Xi ASR \\
f? : FID \\
arr! : TIME \\
\hline
arr! = (flight\, f?).arrTime \\
\hline
\end{array}$$

DepartureTime

$$\begin{array}{|l}
\Xi ASR \\
f? : FID \\
dep! : TIME \\
\hline
dep! = (flight\, f?).depTime \\
\hline
\end{array}$$