XX.
Using Conceptual Graphs to Analyze Multiple Views Of Software Requirements

Harry S. Delugach
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL 35899 U.S.A.

XXX.1. INTRODUCTION

This chapter describes an application of conceptual graphs to support software requirements development — the process of determining what software needs exist and how those needs will be filled. As a human knowledge- and experience-based activity, requirements development is an appropriate domain for applying formal models of cognitive structures. This chapter introduces the following contributions to the theory and practice in conceptual graphs:

a. The ability to represent a conceptual graph that changes over time, using a new class of node called a *demon* node.
b. A structure to partially manipulate informal (external) information (i.e., information not expressed in conceptual graphs), by introducing a special referent form called a *private referent*.
c. The ability to obtain a conceptual graph representation from a requirements specification written in one of several common notations.
d. A framework using conceptual graphs in the analysis of software requirements that effectively captures the overlap between multiple views.

The chapter sections are organized as follows: XXX.2 discusses the general problem of software requirements, for those readers unfamiliar with this aspect of software development. XXX.3 describes two extensions to conceptual graphs that are desirable for capturing requirements. XXX.4 explains how conceptual graphs (as extended) are used to capture requirements. XXX.5 outlines the framework in which multiple views are analyzed. XXX.6 shows some partial results for an example set of requirements.

XXX.7 discusses some issues involved when using conceptual graphs in requirements.

XXX.2. SOFTWARE REQUIREMENTS VIEWS

Software requirements development is the process of determining what is the purpose of a proposed software system. Large-scale software development must satisfy many people and organizations, with differing viewpoints regarding a proposed software system. While many useful requirements development methods have been proposed, they are limited to a particular view, usually the view of the software developer [Alford (1977)], [Alford (1985)], [Teichroew (1977)], [Yourdon (1979)], [Ross (1977)], [Ross (1985)], [Zave (1982)], who must describe all of the requirements using a single notation. Some current methods address the needs of other views, such as an end-user [Wasserman (1982)] or customer [Wartik (1983)]. The latter method also introduces the idea of multiple views in that both a customer and designer's views are considered.

Since organizations and people already have a large investment in current methods (both in their monetary support for tools, and in their understanding of requirements relative to their methods), we would like to incorporate the strengths of their current methods, while at the same time overcoming their limitations.

In this chapter, we describe techniques whereby existing requirements are translated into conceptual graphs, so that several views are represented. In addition, some assumptions underlying each participant's view are expressed in graph form, so that participants' cognitive knowledge is (at least partially) captured for analysis. The resulting set of conceptual graphs can then be analyzed in several ways, such as determining a least common generalization, or performing a join such that a single set of requirements can be obtained.

XXX.2.1. Originating Requirements

Participants may already understand one or more current requirements paradigms. They need not abandon their understanding of the paradigms in order to benefit from using multiple views. Except for the requirements analyst, no one else needs to learn conceptual graphs; each participant starts by expressing his requirements in a language he chooses. Originating requirements are then translated into conceptual graphs for further analysis.

A participant expresses requirements for a target system in a language (*notation*) of his choosing. Once expressed, his set of requirements is called a *requirements specification* (or *R-Spec*). Each R-Spec is therefore identified by (1) a participant's name, (2) a notation, and (3) a target system. An R-Spec is shown in a private referent (see below) whose formal concept is given as

[_ **VP** *p **USING** *n **SYS** *s R-Spec _]

where *p is the participant, *n is the notation, and *s is the target system.

In this chapter, for illustration purposes, we will consider requirements for a patient-monitoring system where each patient is connected to sensors that record his vital signs. When a reading is out of range, an alarm sounds at the nurse's station.

Suppose a customer uses an entity-relationship-attribute diagram to express his requirements. Figure XXX.1 is an R-Spec of the patient-monitoring system requirements from a customer's point of view, using an entity-relationship diagram. Later Figure XXX.1 will be translated into conceptual graphs.

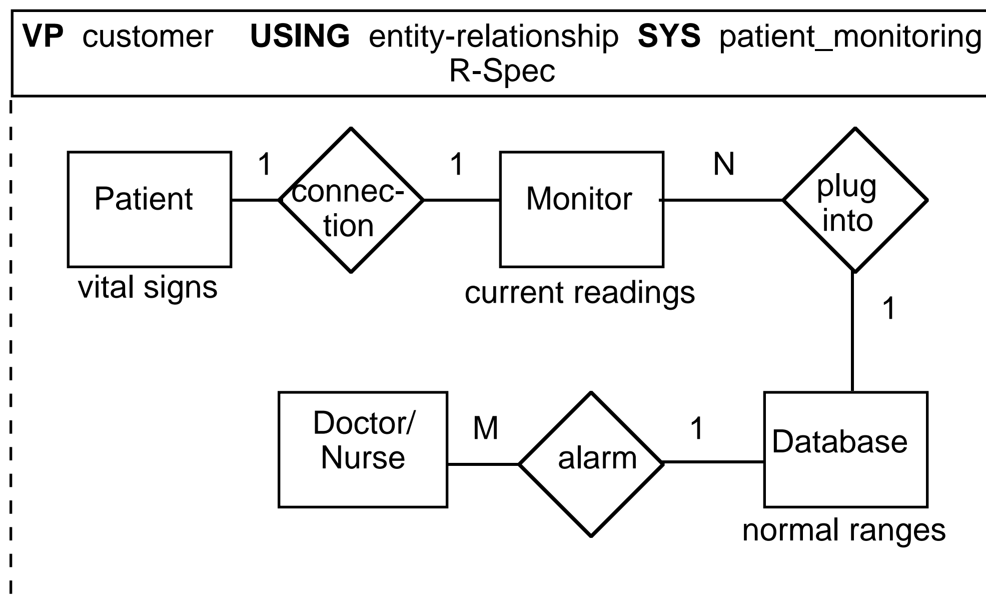


Figure XXX.1.

XXX.2.2. Pre-Existing Assumptions

An important part of multiple-viewed analysis is the use of pre-existing assumptions. These assumptions underlie a participant's R-Spec, yet are too often left implicit during development. For example, a doctor or nurse will assume a patient's vital signs to be a particular set of measurements — tem-

perature, pulse, respiration, and blood pressure. Since using multiple views necessarily involves information outside any one self-contained view, pre-existing assumptions contain requirements information that might not otherwise be available.

This chapter uses such assumptions (expressed as conceptual graphs) to constrain the graphs that represent requirements. Due to the nature of requirements as a people-centered activity, we must acquire these assumptions from human participants directly (i.e., manually). The requirements analyst's task is to discover these assumptions and write down their conceptual graph representations.

Each person involved in developing software requirements is called a *participant*. Typical participants include a customer, designer, end-user, or tester. One special participant — the *requirements analyst*, or simply *analyst* — has the task of coordinating the collection of participants' views and making some judgments about the requirements.

A *view* is defined as one participant's set of requirements, as well as all his underlying assumptions. Since it is impossible to capture every single assumption held by a participant, this definition tells us that our representation of a view will necessarily be incomplete. Even if we could capture a complete view, we would still not have captured all relevant requirements because there are additional participants with other views to consider, as pointed out in [Wartik (1983)], [Wood-Harper (1985)], [Finkelstein (1989)], and [Werkman (1989)].

Any single view will fail to capture the complete set of requirements. Participants have different experience and vocabulary. They need some way to judge the overlap between their views. The purpose of this methodology is to provide a framework to help people explore the consequences and implications of applying such judgments. Some human intervention is required during analysis; after all, it is human beings' requirements that are being sought.

XXX.3. EXTENSIONS TO CONCEPTUAL GRAPHS

Conceptual graphs have the power to express conceptual knowledge about many kinds of requirements; however, since conceptual graphs are based on first-order logic, there are well-known limitations that must be overcome before we can use them for realistic requirements.

Some cognitive structures are not adequately captured by Sowa's notation [Sowa (1984)] with only concepts, relations, and actors. Several variations have been proposed (e.g., [Gardiner (1989)]) that represent some

structures, such as logical implication and sets, more naturally. In many cases, the variations are shorthands for equivalent (but less elegant) representations in the standard notation.

XXX.3.1. Demon: A Node To Represent Temporal Logic

Conceptual graphs' most important limitation with respect to requirements is the lack of explicit representation for temporal knowledge. Although first-order logic with functions is useful for capturing many requirements, there are additional requirements that reflect temporal logic. We therefore need a way to represent conceptual graphs that exist only for a limited period of time. In this chapter, a new conceptual graph node called a *demon* is introduced to express temporal knowledge. Further discussion of demons can be found in [Delugach (1991a)].

Several schemes have been proposed to handle temporal information in conceptual graphs, by identifying certain relations, such as (before), (after), (precedes), etc., whose definitions are carefully coordinated with one another. Recent work has addressed this problem [Esch (1990)], [Moulin (1990)]. I would like to propose a different scheme that involves defining a new kind of node to conceptual graphs, in addition to concepts, relations and actors.

Some useful notions, such as state transition diagrams [Aho (1972)], require a dynamic structure. A new fourth class of node is needed that not only allows alteration of referents, as performed by actors, but also altering the arrangement of concepts over time as well.

The new class of node is called a *demon* in this chapter. Its syntax is similar to an actor's, except that it is enclosed in double angle brackets, as <<transition>> in the linear form, and in the display form as a double diamond.

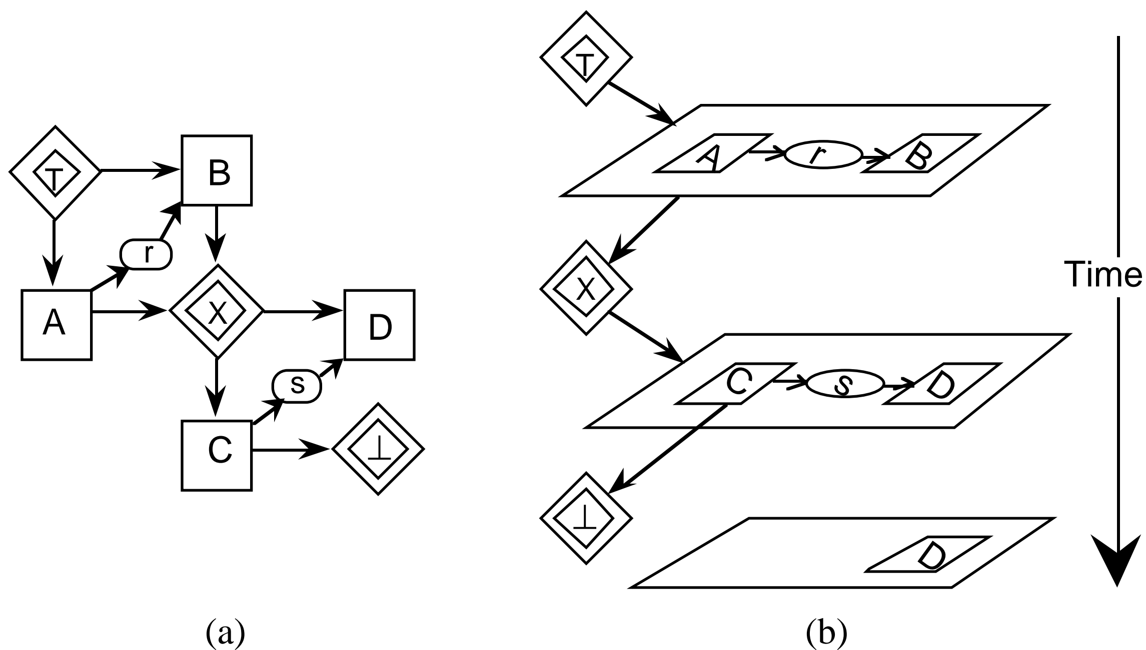
A demon possesses the semantics of an actor node with respect to output concepts' referents, with the additional semantics that a demon actually creates its output concepts (i.e., causes them to come into existence) and then erases its input concepts (retracts them from existence). It therefore represents changes to the arrangement of concepts, etc., on the sheet of assertion.

A demon's input links must all be marked before the demon fires, just as an actor fires. By extension, the creation of a concept can mean the creation of an entire graph. Likewise destruction or retraction can involve entire graphs. What does it mean to create a concept box? Since a concept box represents an existential assertion, creating a concept box on the sheet of

assertion is equivalent to asserting a new proposition. Erasure of a concept is the equivalent of retracting its asserted proposition — i.e., saying, “This fact (or context) is no longer known to be true.” The added semantics allow the expression of dynamic logic, whereby prior truths may be subsequently falsified and new truths introduced. We can thereby express notions of non-monotonic logic, such as discussed in [Turner (1984)].

Two primitive demons are proposed. An *initiator demon* <<⊤>>, possessing only output links, is presumed to fire automatically upon writing down a graph. Therefore all of its output concepts are immediately asserted. A *terminator demon* <<⊥>> possesses only input links; its meaning is that it erases its input concepts (retracts them). As yet, it has no special semantics.

Existing conceptual graph semantics are easily extended for this purpose. A graph containing no demons is assumed to have an initiator demon whose output arcs are connected to every node in the graph. Thus the entire graph is asserted to be true at one time, in the normal sense of conceptual graphs. For example, consider the graph in Figure XXX.2(a):



(a) (b)

Figure XXX.2.

Figure XXX.2(a) can be envisioned as representing a series of different conceptual structures varying over time as shown in Figure XXX.2(b), with each plane representing the arrangement at a given time.

Conceptual graphs with actors are comparable to static allocation in a conventional programming language. Adding the demon’s semantics allows one to represent dynamic allocation, whereby structures may come into being and later pass from existence. Demons are used in this chapter to capture state transition semantics.

Although not considered here, some interesting issues arise regarding demons. An even more powerful demon might be able to change not only referents and arrangements, but also the type hierarchy as well. Further study is needed in these areas. This chapter employs demons primarily to simplify the identification of concepts in state transition diagrams.

XXX.3.2. Private Referent

As others have noted in software development (e.g., [Agresti (1987)], [Zave (1989)]), mental models help us identify new things that fit within our model, but at the same time, those mental models sometimes hinder us from identifying things outside our model.

Even the conceptual graph notation — our mental model for representing other mental models — falls prey to this effect. We must be able to represent some knowledge in the world that is outside of a conceptual graph. This information is shown by a structure we call a *private referent* — an informal referent contained in a formal concept, but considered an un-interpreted (and un-alterable) object. The concept may have one formal referent as its name.

In the display form, a private referent is shown inside a dashed bucket below a formal concept box, as in Figure XXX.3. In the linear form, only its associated type and name are shown within underline-brackets as: [_ Q: n _]; private referents may be difficult to show in linear form, since they do not have to be text.

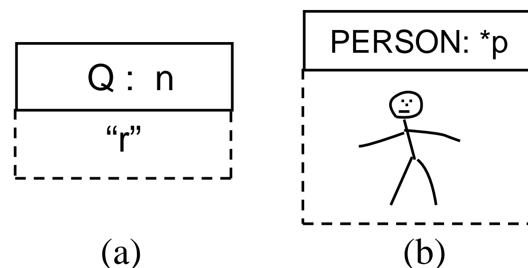


Figure XXX.3.

The meaning of Figure XXX.3(a) is that the solid box [Q: n] is treated as a normal conceptual graph concept, with type Q and name n. The attached private referent "r" is something not represented by conceptual graphs. For example, Figure XXX.3(b) represents the concept [PERSON: *p] with an attached private referent (the little drawing of a person). Note that *p is a formal ("public") referent, able to be changed by normal conceptual graph operations, e.g., an actor's computation.

A private referent cannot be changed, either through generalization, specialization, or the computation of an actor. It is simply carried along with the concept (and its formal referent if present).

Private referents are used in this chapter to handle requirements specifications that are not in conceptual graph form, so that we can still include them in analysis. Private referents can help us refine external information, so that informal knowledge can later be shown (at least partially) in formal conceptual graph notation.

XXX.4. REQUIREMENTS VIEWS IN CONCEPTUAL GRAPH FORM

Analyzing multiple views of requirements begins by translating each participant's originating requirements (R-Spec) into conceptual graph form (an *R-Spec-Graph*). Our purpose will be to combine the R-Spec-Graphs in order to find their common elements.

A *translation scheme* is an algorithm that translates a participant's requirements (R-Spec) into a conceptual graph, called a *requirements specification graph* (or R-Spec-Graph) which is a trial graph for validating multiple-viewed requirements. A translation scheme is specific to a particular notation.

An R-Spec-Graph will be shown as:

[**VP** *p **USING** *n **SYS** *s R-Spec-Graph]

VP indicates the viewer-participant's identity. **USING** denotes the language (or notation) that is used. **SYS** denotes the name of the system that all participants are presumed to be describing. These abbreviations can also be used to denote part of a graph, e.g., [**VP** designer **USING** dataflow **SYS** patient-monitoring PATIENT] means *a patient in the designer's view of a patient-monitoring system using dataflow diagrams*. Later, pre-existing assumptions will be used to further constrain the R-Spec-Graph.

The following briefly summarizes how an R-Spec in each of three models is translated into conceptual graph form as an R-Spec-Graph. Algorithms, additional details and other models, are found in [Delugach (1991b)].

Translating Entity-Relationship Diagrams

Entity-relationship-attribute (ER) diagrams [Chen (1976)] consist of entities (shown by boxes), each with zero or more associated attributes, where an entity is connected to one or more other entities by a relation (shown by a diamond). Others have already studied this model [Muehlbacher

(1989)], [Muehlbacher (1990)], [Gray (1991)]. Since this paradigm is similar to conceptual graphs, we adopt these rules for translating its structure:

- An ER entity *E* becomes a conceptual graph concept [ENTITY: E].
- An ER attribute *A* associated with entity *E* becomes the sub-graph
  [E] → (attribute) → [A].
- An ER relation *r* becomes a conceptual graph relation (r).

Additional rules determine the cardinality of each referent, based on the cardinality of the original ER relation. Using the R-Spec shown in Figure XXX.1 above, we obtain the R-Spec-Graph in Figure XXX.4:

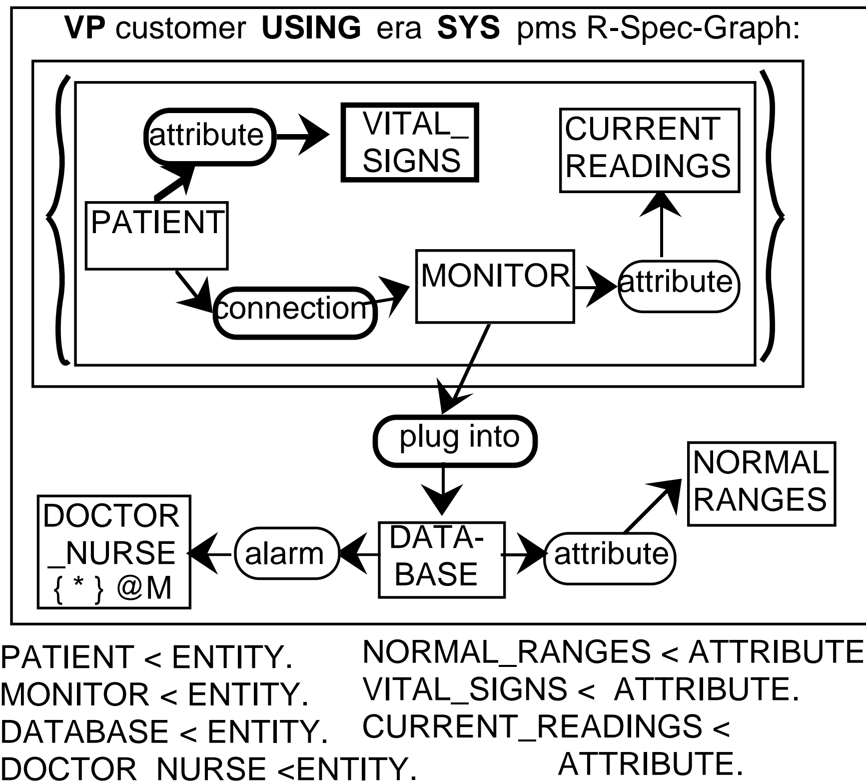


Figure XXX.4.

Translating Data Flow Diagrams

A data flow diagram (e.g., [Yourdon (1979)]) consists of process nodes connected by labeled directed arrows indicating the flow of data [Sowa (1984)]. Suppose a data base designer uses data flow diagrams to express his requirements. An example data flow R-Spec is shown in Figure XXX.5.

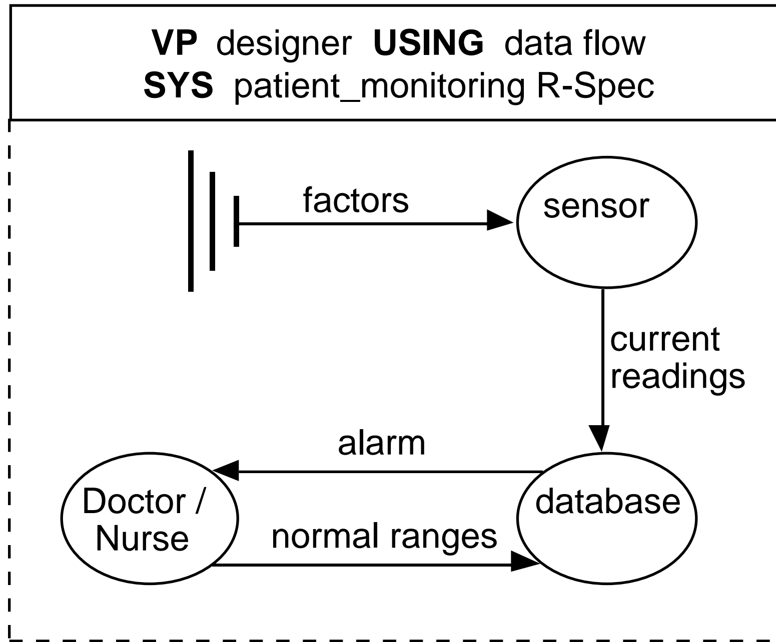


Figure XXX.5.

Our structural translation rules are summarized as follows:

- A data flow process *P* becomes a conceptual graph actor <P>.
- A data flow arc from process *A* to *B* with label *L* becomes a conceptual graph concept of type *L* linked to actor <B> and from actor <A>.

Additional rules handle sources and sinks. Figure XXX.6 shows the translated R-Spec as an R-Spec-Graph.

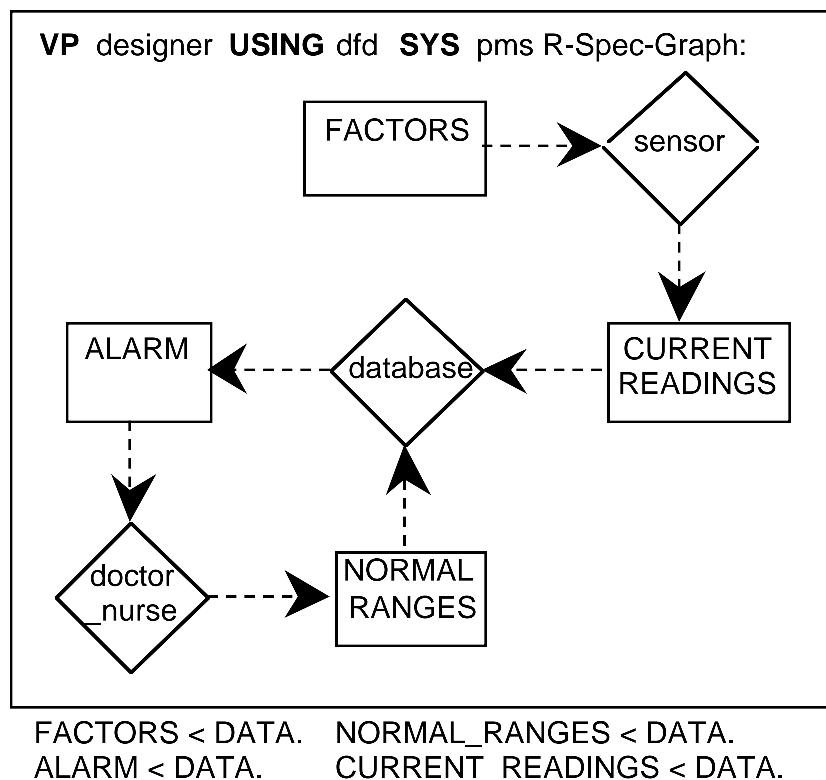


Figure XXX.6.

Since this graph contains no relations, we will use a special dependency rule that relates actors to corresponding concepts.

Translating State Transition Diagrams

State transition diagrams consist of a set of states, input events, output events, a start state and a set of final states [Aho (1972)]. A state transition function determines a new state and output event for every possible current state and input event. Suppose an end-user (doctor or nurse) uses state transition diagrams to express his requirements. Figure XXX.7 shows an example state transition R-Spec:

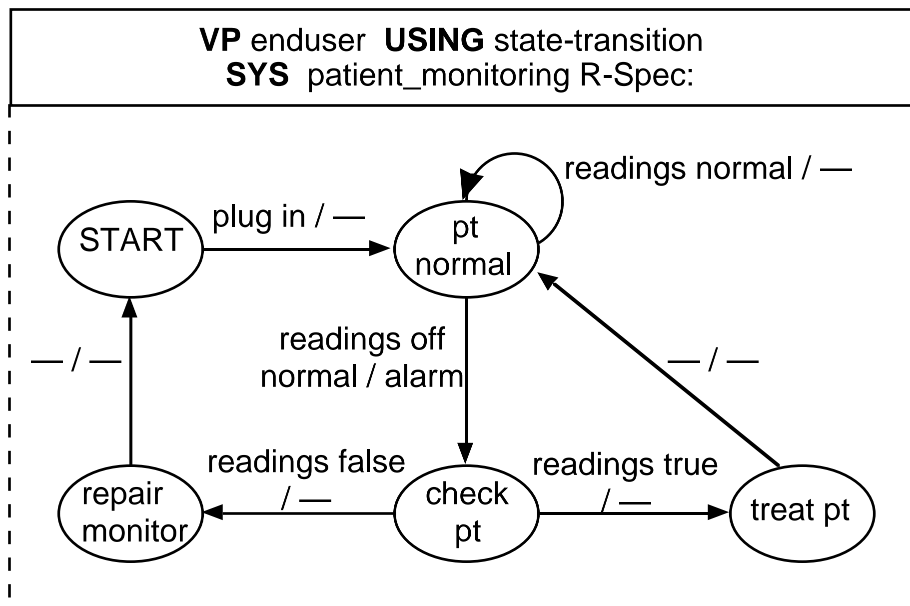


Figure XXX.7.

Using our new demon node, our structural rules are as follows:

- A state *S* becomes a conceptual graph concept [S].
  S < STATE is added to the type hierarchy.
- An input or output event *J* becomes a conceptual graph concept [J].
  J < EVENT is added to the type hierarchy..
- A final state *S* is denoted by attaching the monadic relation (final) as:
  (final) —> [S].
- A transition from state *S* with input *J* to state *T* with output *K* becomes a demon with links to [T] and [K], and with links from [S] and [J]. Add K < EVENT. T < STATE to the type hierarchy.

The translated R-Spec-Graph is shown in Figure XXX.8:

VP enduser **USING** std **SYS** pms R-Spec-Graph:

PLUG_IN < EVENT.
ALARM < EVENT
START < STATE.
NORMAL < STATE.
CHECK_PT < STATE.
TREATMENT < STATE.
READINGS_NORMAL < EVENT.
READINGS_FALSE < EVENT.
READINGS_OFF_NORMAL< EVENT.
READINGS_TRUE < EVENT.
REPAIR_MONITOR < STATE.

Figure XXX.8.

The transition demon's semantics (see §XXX.3 above) ensure that an old state is retracted when a new state comes into existence.

Once separate R-Spec-Graphs have been obtained, we will no longer refer to the originating requirements (R-Specs); instead, we use their R-Spec-Graphs.

Assumption Graphs

One problem with any specification is that there are implicit underlying assumptions which affect its meaning. Naturally we want to capture such assumptions in conceptual graph form, called an *A-Graph.* Some examples of underlying assumptions are:

1. Customer assumes that when an alarm sounds, doctors and nurses start treatment after a certain period of response time.

2. Designer assumes database is accessed through a query/process/response cycle.
3. Doctor/nurse assumes that current readings are either within range or out of range.

Assumption 1 is expressed the conceptual graph in Figure XXX.9. Assumption 2 is partly expressed by the conceptual graph shown in Figure XXX.10. Assumption 3 is expressed by the graph in Figure XXX.11.

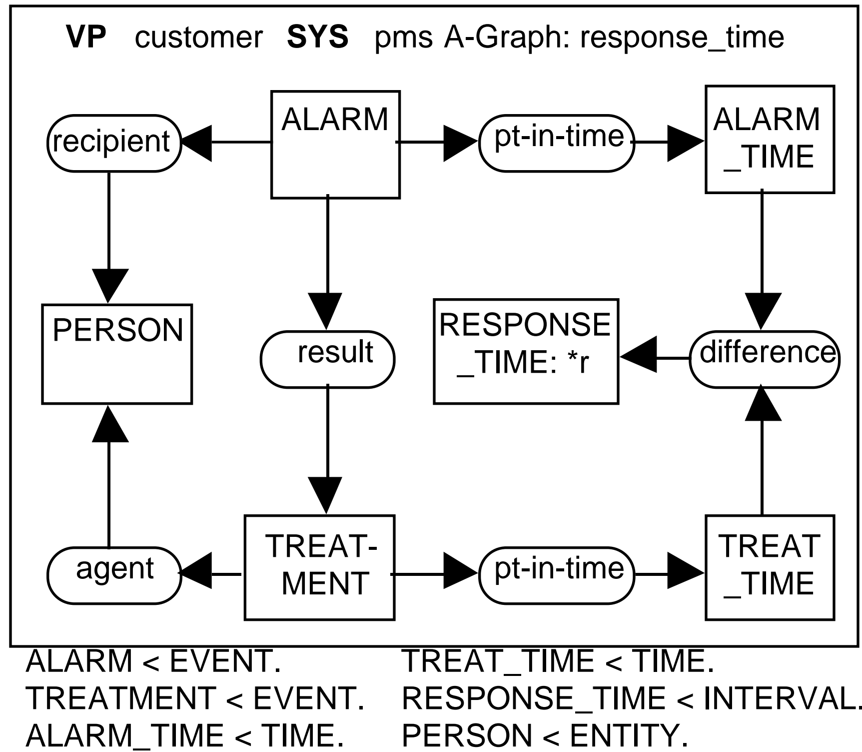


Figure XXX.9.

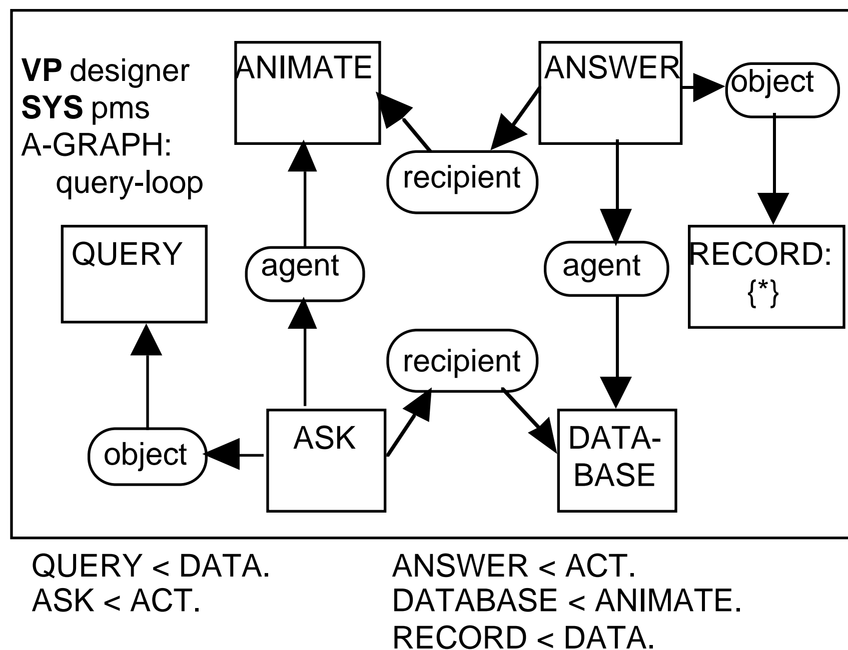


Figure XXX.10.

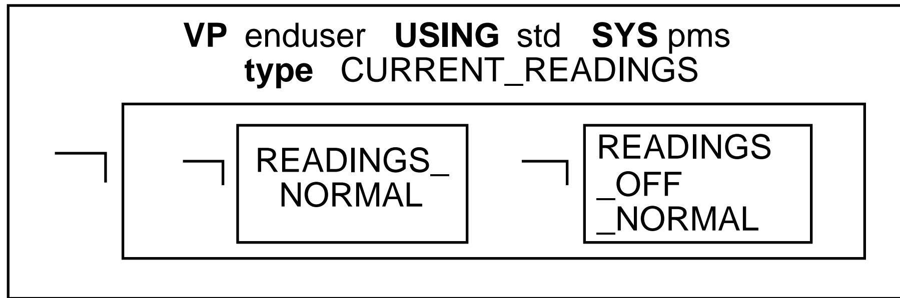


Figure XXX.11.

If these example assumptions seem simple or obvious, note that that our biggest problem with implicit assumptions is not their complexity — it is that they are hidden. Even simple assumptions, when left implicit, may not be known by all participants.

The process of applying an assumption may just mean asserting it onto the sheet of assertion. Some assumptions are thus "thrown into the pot," so to speak; once identified, they act as definitions for that view's conceptual graph context. More often, however, we wish to join an assumption to an R-Spec-Graph, or another assumption in a different view.

XXX.5. ANALYZING MULTIPLE VIEWS OF REQUIREMENTS

Our next step is to identify information that appears in more than one view. We use two primitive notions – counterpart and dependency – on which to base our analysis.

XXX.5.1. Counterparts Between Views

In order to analyze a combination of requirements views, there must be at least one common point of reference between them. We discuss two kinds of common points: concept-to-concept correspondence and concept-to-actor correspondence.

Concept-to-Concept Counterparts

A useful point of reference is to consider individuals in common between two or more R-Spec-Graphs. We call any two such individuals *counterparts* to each other, in that they represent the same real-world (or imagined-world) individual. In this chapter, a counterpart is restricted to being either a concept box in one or more views, or an actor-concept pair. A counterpart is a reflexive, symmetric, and transitive relation.

Counterparts are a reasonable starting point because two participants might see the same individual as members of different classes. For example

a family member might see John as the concept [FATHER: John] where his doctor might see him as [PATIENT: John]. Conceptual graphs provide a convenient way of relating both concepts through a type hierarchy — their least common generalization is [PERSON: John]. A least common generalization of [T: John] would indicate that *something* named John exists in both views. That fact alone may not be useful in itself, but it may lead the analyst to gather more information (e.g., seek additional assumptions or refine one's originating requirements) to determine whether the Johns are actual counterparts. If the least common generalization is just [T] (i.e., [T: *]), however, we cannot say they are counterparts.

A generic concept stands for some arbitrary individual belonging to a certain class; in the absence of a specific referent, assume "*". For example, in the graphs above, there are three counterpart-pairs:

[**VP** customer **VP** designer NORMAL_RANGES: *]
[**VP** customer **VP** designer CURRENT_READINGS: *]
[**VP** designer **VP** end-user ALARM : *]

Once counterparts are identified in two or more R-Spec-Graphs, the graphs can be joined around the counterparts into a single graph, effectively combining multiple sets of requirements into one. The resulting combined graph can then be analyzed according to its own internal consistency, as well as having pre-existing assumptions (possibly from other multiple views) applied to it.

In Figure XXX.9, alarm is shown as an individual, whereas our counterpart is

[**VP** designer **VP** end-user ALARM]

with its type defined as ALARM < EVENT.

Our definition of a counterpart allows the analyst to decide that [**VP** customer EVENT: alarm] is a counterpart to the already-identified counterparts, since alarm is an individual's name in **VP** customer. Thus we have added to the meaning of alarm by determining its counterpart in all three views:

[**VP** customer **VP** designer **VP** end-user EVENT: alarm]

While some counterparts can be identified by examining results of an automatic generalization (given an appropriate type hierarchy), some counterparts cannot. Consider the case where the same individual may have different names in different views. An automated method must examine every possible concept-concept pair to determine some correspondence between

them. As an example, one participant may call an analog device a "sensor", while the other calls it a "monitor". One R-Spec-Graph might contain

[SENSOR] → (connection) → [PATIENT]

while the other might contain

[MONITOR] → (connection) → [PATIENT]

If [PATIENT] were already identified as a counterpart, we would have evidence for establishing [SENSOR] and [MONITOR] as a counterpart, but since in this case their least common generalization is [T], only human participants can actually decide.

Two individuals, each in different views, may have the same name, but are not the same individual. An automatic method may identify an inconsistency — i.e., some fact about one that is not known to be true about the other — but human intervention must decide whether the inconsistency really exists or is simply a feature of the counterparts being in different views. For example, if the following two relations occurred in two different R-Spec-Graphs,

[MONITOR] —> (connection) —> [DATABASE]
[MONITOR] —> (connection) —> [PATIENT]

the two monitor definitions would be inconsistent (see below), since the minimal supertype of DATABASE and PATIENT is just "T". We discuss this situation below when we consider issues of consistency among views.

XXX.5.2. Dependency

A dependency between views is when there is a concept in one view whose existence is dependent upon a counterpart in another view. Dependency is intended in the sense of Schank's conceptual dependency, e.g., in [Schank (1975)], where the existence of one concept implies the existence of another. Dependency is thus reflexive and transitive, but it is not symmetric; i.e., if A depends on B, B will not in general depend on A.

Rules for identifying dependency among counterparts may involve information external to a participant's model, but the knowledge may be nonetheless useful in increasing understanding. Some examples of dependency rules are:

1. The output of a process depends upon its input.
2. In an ER diagram, an attribute depends upon the entity possessing it.
3. An actor *C* originating from a data flow diagram depends upon an entity *C* originating from an ER diagram, if such an entity exists.

The first rule is implemented by actors, in that an actor's output referents depend upon the result of computation using its input referents. The second rule is enforced by the links' direction in the relation (attribute). The third rule is implemented by the concept-actor rule (see XXX.4 above).

Dependency is shown by a double arrow as in [A] => [B] meaning "B depends upon A". A relation's dependency may be kept as part of its definition; e.g., the definition of (attribute):

```
relation attribute is
    [ ENTITY: *ent ] —> (attribute) —> [ ATTRIBUTE: *att ];
        *ent => *att.
```

XXX.5.3. Consistency

We can identify three kinds of consistency to be determined for multiple viewed requirements graphs, called strong consistency, weak consistency and strong inconsistency, which are explained below.

Strong Consistency

A strong consistency between counterparts is where a counterpart means the same thing in both views. We consider a strong consistency as a fact about one view that is provable by facts in another view. This amounts to having a concept in one view possessing the same links and relations as its counterpart in another view.

Weak Consistency

A weak consistency between views is where a counterpart means different things (i.e., has different relations and is connected to different concept types) in different views. Inconsistency is defined as where some fact about one counterpart in one view cannot be proven or disproven by facts in another view.

Weak consistency does not necessarily indicate a fault in the requirements. It may merely reflect the incompleteness or limitations of one view relative to the others. For example, if [PATIENT] is a counterpart, one view might contain:

[**VP** A   PATIENT] → (attribute) → [VITAL_SIGNS]

whereas another view might contain:

[**VP** B   PATIENT] → (connection) → [SENSOR]

These two graphs provide two orthogonal constraints on [PATIENT]: neither one is provable from the other, nor can one be disproven by the other.

Strong Inconsistency (Conflict)

A conflict between views is a strong inconsistency, where a counterpart in one view is incompatible with constraints in another view. Conflict is defined as where some fact about one counterpart in one view can be proven false in another view. For example, if [**VP** A PERSON: p] and [**VP** C EVENT: p] both occur, then at least one of them must be incorrect, since their least common generalization is the type "T".

Identifying conflicts is an important part of understanding requirements. Once conflict are identified, however, participants may use that knowledge in manual or heuristic ways in order to make appropriate changes to their originating requirements.

XXX.5.4. Completeness

Completeness is easier to understand if we consider its inverse: incompleteness. Two forms of incompleteness may be discussed. One form of incompleteness results when existing requirements lack features than can be inferred from other existing features or pre-existing assumptions. Another form of incompleteness results from a participant failing to express one or more of his needs.

Our previous analysis addressed the first form of incompleteness. We noted that a seeming weak inconsistency may merely reflect the known fact that each view is incomplete. The second form, however, is a fundamental human limitation — although using multiple views may improve a participant's insight or judgment, completeness is more a goal to guide and inspire us than an achievable result.

Once all analysis has been performed, the overlap between two views can be characterized by the combined graph, a set of counterparts, a set of dependencies, and the results of the three kinds of consistency checks.

XXX.6. EXAMPLE OF MULTIPLE VIEWED REQUIREMENTS

Previous sections showed the graphs making up each view of a patient monitoring system's requirements. This section shows the summarized results of analyzing their combination. Refer to Figure XXX.13 for their

display forms. The top part of the picture is the analyst's R-Spec-Graph containing the common elements from the three views.

Counterparts in Patient-Monitoring. Counterparts are shown in Figure XXX.13 as being connected via the (counterpart) relation. We could show the three views' graphs as being joined around the counterparts; however, the counterpart relations are retained to indicate their derivation.

Dependencies in Patient-Monitoring. The dependencies in the patient-monitoring example are shown in Figure XXX.12 below. Some of the dependencies were obtained from relation definitions not shown in this chapter, but were a part of the analyst's set of definitions for multiple viewed analysis.

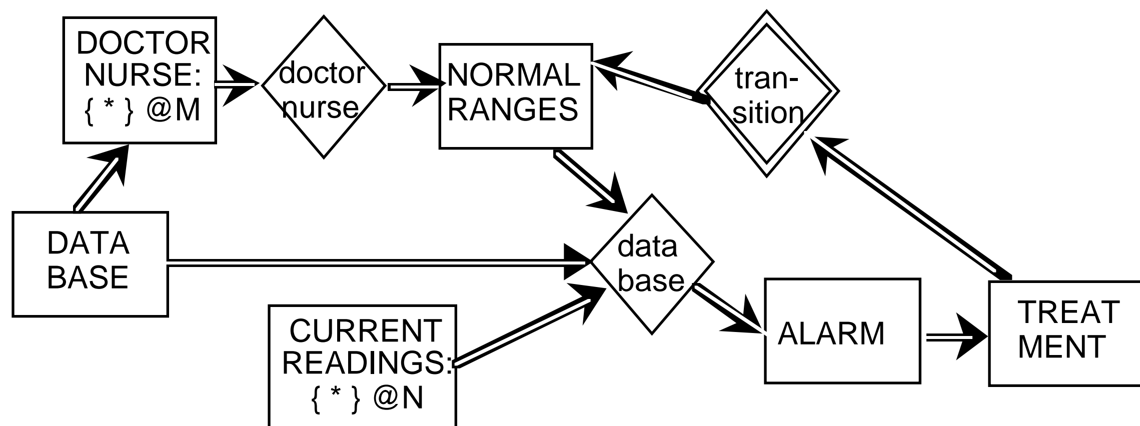


Figure XXX.12.

Least Common Generalization. The least common generalization between the views is represented by the analyst's R-Spec-Graph, shown in the top portion of Figure XXX.13.

Joined Graphs. The combined requirements of all three participants — customer, designer, and end-user — are shown in Figure XXX.13. Note how the assumption graphs contribute to the complete set of requirements.

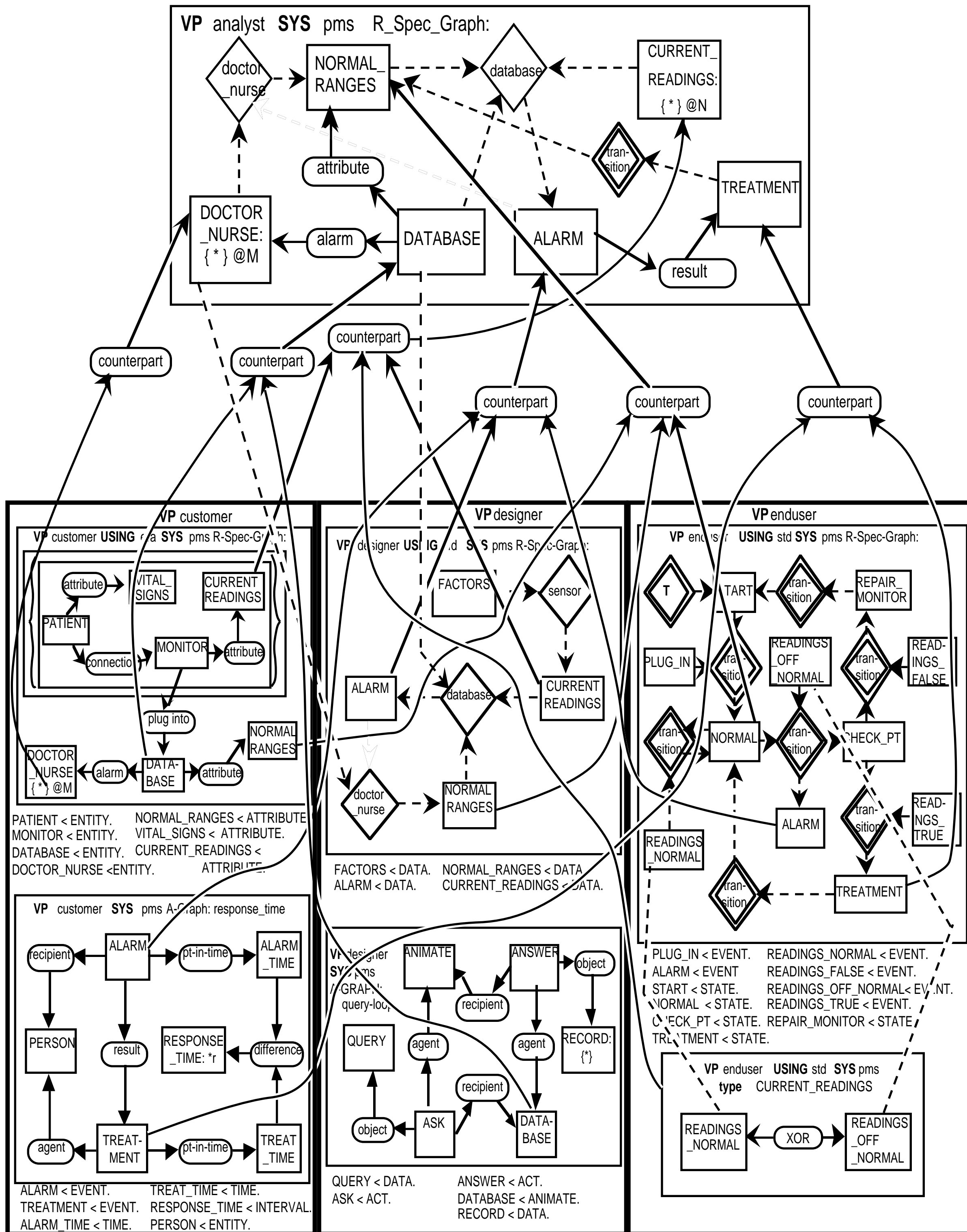


Figure XXX.13.

XXX.7. ISSUES IN MULTIPLE VIEWED REQUIREMENTS

Many interesting issues need to be addressed in order for conceptual graphs to be a viable tool for multiple viewed requirements development. This section outlines a few of the issues involving conceptual graphs; see [Delugach (1991b)] for more discussion.

Validating Extensions To Conceptual Graphs. Like any extension, the private referent, and the demon node need to be explored more thoroughly to validate their properties with respect to our accepted meaning of conceptual graphs.

Presenting Results. Some results we want to present to a participant are not expressible in a participant's original view, because it was obtained from other views. Choosing a language in which to present the results is not an easy task. We can easily display a combined graph; however, unless all participants learn conceptual graphs, the combined graph will be of little use.

Since conceptual graphs can be expressed in natural language, if we capture results in conceptual graph form, we may use English paraphrases for presenting our findings to multiple participants; however, paraphrasing remains a difficult operation (see [Sowa (1983)]).

Acquiring Pre-Existing Assumptions. A key part of using multiple views is the acquisition of a participant's pre-existing assumptions. This task is still almost entirely heuristic, based on the requirements analyst's intuition and experience. We need automated methods to identify and formalize these assumptions. Perhaps we can involve a participant in an interactive dialogue that can automatically discover some assumptions.

Formal Meaning Of Overlap. Discussing multiple views is difficult because to each participant, any other view's information appears outside of his own view. Although informal knowledge of other views may be useful, we would like to know if there is some logical basis for including such external knowledge in a participant's view. How does feedback occur when knowledge of overlap results in a participant changing his original requirements?

Granularity and Computability. This work considers as counterparts only those concept boxes that represent the same individuals. We have made no provision for the case where a sub-graph has a counterpart. This is because to examine all sub-graphs is computationally expensive. If two views use substantially different levels of detail, then a sub-graph's counterpart might possibly be another sub-graph of any size. Without the use of constraining pre-existing assumptions, it is unlikely that differing degrees of granularity can be overcome for large sets of requirements.

A different kind of granularity involves looking at differing degrees of "counterpart-ness," where we have the same individual but at different times, or the same individual in different places. For example, the dependency

relationship between an actor and a like-named concept(s) is akin to a counterpart.

Summary

Conceptual graphs are a valuable language for capturing multiple views of software requirements. We have outlined a framework in which conceptual graphs provide a basis for analyzing multiple views, and their underlying assumptions. Overlap between views can be characterized as a set of counterparts and dependencies, along with strong and weak inconsistencies. Conceptual graphs are a promising avenue for gaining insight into the problems of multiple viewed requirements.

REFERENCES

Agresti, William W. (1987) "Guidelines for Applying the Composite Specification Model," Software Engineering Laboratory, Tech. Report no. SEL-87-003, Greenbelt, MD, U.S.A.

Aho, Alfred V. (1972) *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Vol. I, Englewood Cliffs, NJ.

Alford, Mack W. (1977) "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Trans. Softw. Eng.*, Vol. SE-3, no. 1, pp. 60-69.

Alford, Mack W. (1985) "SREM at the Age of Eight: The Distributed Computing Design System," *IEEE Computer*, Vol. 18, no. 4, pp. 36-46.

Chen, Peter Pin-Shan (1976) "The entity-relationship model — toward a unified view of data," *ACM Trans. Database Sys.*, Vol. 1, no. 1, pp. 9-36.

Delugach, Harry S. (1991a) "Dynamic Assertion and Retraction of Conceptual Graphs," *Proc. 6th Annual Workshop on Conceptual Graphs*, Way, Eileen C., ed., pp. 15-26, SUNY Binghamton, Binghamton, New York.

Delugach, Harry S. (1991b) "*A Multiple-Viewed Approach to Software Requirements*," Ph.D. thesis, Computer Science Dept. University of Virginia, Charlottesville, Va.

Esch, John and Timothy Nagle (1990) "Representing Temporal Intervals Using Conceptual Graphs," *Proc. of the 5th Annual Workshop on Conceptual Structures*, Eklund, Peter and Laurie Gerholz, eds., pp. 43-52, Linköping University, Boston & Stockholm.

Finkelstein, Anthony and Hugo Fuks (1989) "Multiparty Specification," *Proc. 5th Intl. Wkshop, Softw. Spec. and Design*, pp. 185-195, ACM SIGSOFT.

Gardiner, David A., Bosco Tjan, and James R. Slagle (1989) "Extended Conceptual Structures Notation," *Proc. 4th Annual Workshop on Conceptual Structures*, Nagle, Janice A. and Timothy E. Nagle, eds., pp. 3.05, AAAI - IJCAI-89, Menlo Park, CA 94025.

Gray, Linda C. and Ronald D. Bonnell (1991) "A Comprehensive Conceptual Analysis Using ER and Conceptual Graphs," *Proc. 6th Annual Workshop on Conceptual Graphs*, Way, Eileen C., ed., pp. 83-96, SUNY Binghamton, Binghamton, New York.

Moulin, Bernard and Daniel Côté (1990) "Extending the Conceptual Graph Model for Differentiating Temporal and Non-Temporal Knowledge," *Proc. of the 5th Annual Workshop on Conceptual Structures*, Eklund, Peter and Laurie Gerholz, eds., pp. 105-116, Linköping University, Boston & Stockholm.

Muehlbacher, Robert (1989) "A Conceptual Graph Based Dictionary as a Source for the Generation of Entity Relationship Models," *Proc. 4th Annual Workshop on Conceptual Structures*, Nagle, Janice A. and Timothy E. Nagle, eds., pp. 3.8, AAAI - IJCAI-89, Menlo Park, CA 94025.

Muehlbacher, Robert (1990) "Using Conceptual Graphs as a Representation Language for System Analysis Methods," *Proc. of the 5th Annual Workshop on Conceptual Structures*, Eklund, Peter and Laurie Gerholz, eds., pp. 221-232, Linköping University, Boston & Stockholm.

Ross, Douglas T. and Jr. Kenneth E. Schoman (Jan. 1977) "Structured Analysis for Requirements Definition," *IEEE Trans. Softw. Eng*, Vol. SE-3, no. 1, pp. 6-15.

Ross, Douglas T. (1985) "Applications and Extensions of SADT," *IEEE Computer*, Vol. 18, no. 4, pp. 25-35.

Schank, Roger C. (1975) *Conceptual Information Processing*, Elsevier Science Publ.

Sowa, John F. (1983) "Generating Language from Conceptual Graphs," *Comp. Math. with Applications*, Vol. 8.

Sowa, John F. (1984) *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publ. Co., Reading, Mass. U.S.A.

Teichroew, Daniel and Ernest A. Hershey, III (1977) "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Softw. Eng.*, Vol. SE-3, no. 1, pp. 41-48.

Turner, Raymond (1984) *Logics for Artificial Intelligence*, John Wiley & Sons, New York.

Wartik, Steven P. (1983) "*A Multi-Level Approach to the Production of Requirements for Interactive Computer Systems*," Ph.D. thesis, University of California, Santa Barbara.

Wasserman, Anthony I. and David T. Shewmake (1982) "Rapid Prototyping of Interactive Information Systems," *ACM SIGSOFT Softw. Eng. Notes*, Vol. 7, no. 5, pp. 171-180.

Werkman, Keith J. and Donald K. Hillman (1989) "Designer Fabricator Interpreter System: Using Conceptual Graphs To Represent Perspectives Between Cooperating Agents," *Proc. 4th Annual Workshop on Conceptual Structures*, Nagle, Janice A. and Timothy E. Nagle, eds., pp. 4.14, AAAI - IJCAI-89, Menlo Park, CA 94025.

Wood-Harper, A. T., Lyn Antil, and D. E. Avison (1985) *Information Systems Definition: The Multiview Approach*, Blackwell Scientific Publ., Oxford, U.K.

Yourdon, Edward and Larry L. Constantine (1979) *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, NJ.

Zave, Pamela (1982) "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Trans. Softw. Eng.*, Vol. SE-8, no. 3, pp. 250-269.

Zave, Pamela (1989) "A Compositional Approach to Multiparadigm Programming," *IEEE Software*, Vol. 6, no. 5, pp. 15-25.