Specifying Multiple-Viewed Software Requirements With Conceptual Graphs

Harry S. Delugach

Department of Computer Science
Computer Science Building
University of Alabama in Huntsville
Huntsville, AL 35899
Phone: (205) 895-6614
Fax: (205) 895-6239
Electronic-mail: delugach@cs.uah.edu

running title:
Multiple-Viewed Software Requirements

ABSTRACT

Among all the phases of software development, requirements are particularly difficult to specify and analyze, since requirements for any large software system originate with many different persons. Each person's view of the software requirements may be expressed in a different notation, based on that person's knowledge, experience, and vocabulary. In order to perform a knowledge-based analysis of the requirements in combination, a single knowledge representation must be capable of capturing the information expressible in several existing requirements notations. This paper introduces the notation of conceptual graphs based on semantic networks, that provides a general representation. Four common requirements notations are shown to be expressible using conceptual graphs; with algorithms and examples provided.

ABOUT THE AUTHOR

Dr. Harry S. Delugach is an Assistant Professor in the Computer Science Department, University of Alabama in Huntsville. He held an appointment as a Visiting Assistant Professor in Mathematical Sciences at Memphis State University. He was a programmer in the College of Business at the University of Tennessee, where he also was the chief technical consultant managing U.S. Census data for the Tennessee State Planning Office. He was a computing analyst for the Department of Energy implementing a distributed materials management system in Oak Ridge, Tennessee. He holds degrees from Carleton College (Northfield, Minnesota), the University of Tennessee and received his Ph.D. in Computer Science from the University of Virginia in 1991. His research interests include software engineering, software requirements, logic programming, knowledge representation, and computer science education. He was inducted into the national engineering honor society Tau Beta Pi (VA A) in 1987, and is a member of the ACM and the IEEE Computer Society.

1. INTRODUCTION

Among all the phases of software development, requirements are particularly difficult to specify and analyze, as summarized in [1]. Unlike later phases of software development, the requirements phase cannot be characterized as an effort to preserve decisions made in an earlier step: there *are* no earlier steps. Requirements are based upon human needs, experience, and knowledge.

Most current approaches to software requirements espouse some particular paradigm that is meant to capture all relevent aspects of a proposed software system. Usually this paradigm is embodied in a single requirements language, such the Software Requirements Engineering Methodology's (SREM) RSL [2] [3], SADT [4] [5], CORE diagrams [6], RML [7], entity-dataflow diagrams [8], or statecharts [9]. While these languages have been carefully designed to suit the purposes of requirements development, two obstacles restrict their usefulness: (1) each language's limitations make it inconvenient or impossible to express some important requirements, and (2) software developers already have a substantial investment in their currently-used methods — both in an economic and a cognitive sense — that they are reluctant to give up.

Recently multiple-viewed approaches have been proposed, such as Software Through Pictures [10], DARWIN [11], the Composite Specification Method [12], and the multiparty approach [13]. Their main drawback is the lack of a formal basis from which to automatically analyze the resulting multiple-viewed requirements. This work attempts to overcome that lack in existing methods, while still supporting their use.

Existing methods have inherent limitations. Software Through Pictures [10] permits different parts of a system to be specified in different notations, but their overlap is not formally derived. Like-named objects are assumed to represent the same instance, just as unlike-named objects are assumed to represent different instances. Only a few specific kinds of overlap are pre-defined (e.g., an object being decomposable into a data structure diagram) which limits the amount of semantic information that can be used for analysis.

Darwin [11] uses two similar Ada-like representations for two views: a customer and end-user, but there is no algorithmic technique for translating one into the other, nor is there support for including any additional views.

The Composite Specification Method (CSM) [12] allows three different viewpoints to be represented. Its use supports the claim that multiple views increase understanding of software requirements. There are, however, no formal methods given for transforming between views or for combining views. Also, the three viewpoints are pre-determined; no additional viewpoints are easily included.

The multiparty specification method [13] presents a formal model of the dialogue between viewpoints. Although its motivation is similar to this work's, its authors acknowledge that the knowledge in each viewpoint is not yet formally represented.

This paper presents a general framework for capturing knowledge in several requirements viewpoints, and gives methods for formally transforming different views to and from a general representation and the respective views. The framework is extensible in that additional views can be incorporated without changes to pre-existing transformation methods.

The paper is organized as follows. Section 2 gives the rationale for using a single internal representation, then introduces conceptual graphs as a general knowledge representation for software requirements. Sections 3 through 6 show how conceptual graphs capture information in each of four common requirements notations. Section 7 provides a brief summary. The Appendix gives detailed algorithms for each of the four translations.

2. REPRESENTING REQUIREMENTS KNOWLEDGE

This paper is part of an ongoing effort to accommodate multiple views of requirements so that existing familiar methods can still be employed by developers who want to use them. The job of the requirements analyst is to incorporate several requirements specifications into the analysis process and thereby discover overlapping and conflicting requirements that would not otherwise be visible. To do this, we must first develop a formal framework in which we can capture several different requirements languages.

The framework requires an internal common representation, one which is capable of capturing knowledge from several different requirements languages. It is not our intention that the analyst use this common representation as a requirements language; rather we intend for requirements specifications in existing requirements languages to be translated to and from the internal form. Developers can then view requirements externally using different requirements languages for different purposes.

The reasons for using a single common representation for accommodating multiple views are two-fold. The first reason is that a common internal representation reduces the number of translation algorithms that must be developed. Consider a framework without a common representation, as illustrated in Figure 1(a). With just four representation languages, six two-way translations are necessary if we are to present requirements in any view desired. Adding a fifth representation language would require creating an additional four two-way translations: one for each of the existing languages. Such a framework discourages the inclusion of new requirements languages. With a common representation, however, as illustrated in Figure 1(b), there are only four two-way translation algorithms; moreover, each additional language involves creating just one two-way translation between it and the common representation.

The second reason for using a common internal representation is that analysis can be performed on all the requirements in combination, without omitting any information that might necessarily be lost if analysis were performed using any one of the external requriements languages. Although we do not describe the analysis here, in future work we will explore analysis techniques that operate on the internal representations. Of course, it is crucial that we choose a common representation that can capture information contained in many requirements languages.

Conceptual Graphs

Since requirements originate with human beings, it is reasonable to base our framework around a representation that models mental structures. Toward that end, some recent approaches are based on requirements' cognitive aspects [7], [13]. This paper proposes using the representation of conceptual graphs [14], an extension of semantic networks [15], as a basis for supporting a multiple-viewed framework. Conceptual graphs are a knowledge representation form that has been the subject of several recent workshops [16] [17] [18].

Conceptual graphs are composed of nodes and directed arcs. There are four types of nodes:

(*i*) *concepts* (shown as rectangles) representing individuals, containing a type (class) identifier and a referent showing individual names and cardinality;

(*ii*) *relations* (shown as circles or ovals) representing relationships between them, with arc directions denoting their conceptual dependency;

(*iii*) *actors* (shown as diamonds) representing processes that can change the referents of their output concepts, based on their input concepts;

(*iv*) a new type of node called a *demon* (shown as a double diamond) that causes creation and retraction of its input and output concept(s) respectively.

Figure 2(a) is a conceptual graph representing the sentence: "Pressure is caused by some gas, and is measured in atmospheres." Conceptual graphs can be given in a graphical form, called the *display form*, or as text, called the *linear form*. For example, a concept is shown as a box in display form, or within square brackets (e.g., [PERSON]) in the linear form. An entire graph may be drawn within a context box that can serve as a single higher-level concept. A concept or an entire context may be negated. Definitions (called *canonical graphs*) constrain arrangements of concepts and relations to just those that are meaningful. For example, the canonical graph in Figure 2(b) indicates that an act operates upon a concept of type ENTITY and is caused by some ANIMATE type. A canonical graph has the same components as a simple graph; by being considered part of a *canon* (i.e., a previously-defined set of concepts and relationships), such a canonical graph has the additional semantics that it is assumed to be true.

A concept box contains a type identifier and a referent field. A type identifier represents a class of concepts. Type names are arranged within a hierarchy, so that A < B means that type A is subtype to B. For example, PERSON < ANIMATE indicates that the class of persons is a subtype of the class of animate concepts. The referent field allows specification of individual instances of the concept. For example, the concept [DOCTOR: Jones] identifies one particular named instance; the concept [DOCTOR: *d] denotes a generic instance which is given a symbolic name "d"; the concept [DOCTOR: { * }@3] denotes a set or aggregate instance of three (un-named) doctors; and [DOCTOR: { * }] denotes a set of some unspecified cardinality consisting of doctors.

Two or more concepts that represent the same instance or occurrence are called *co-referents*. Such concepts can be connected directly by a dashed line called a *line of identity* or a *co-referent link*. For example, the graphs in Figure 2(b) represent the sentence: "Doctor Jones is 45 years of age and he is the agent of an act of surgery performed on a patient."

This paper explains how we translate originating requirements specifications (called *R-Specs*) into equivalent conceptual graphs (called *R-Spec-Graphs*). Our purpose is two-fold: first, since we claim that conceptual graphs are more general than current specification methods, we want to show how conceptual graphs capture a few of them, while familiarizing the reader with the notation. Second, since we want in the future to use multiple requirements specifications developed under existing methods, we must show that we can faithfully translate them into conceptual graph form, in order to analyze each specification with respect to the others.

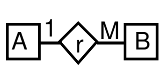
Translation Schemes

The two algorithms that translate a diagram into conceptual graphs and back are called a *translation scheme*. One algorithm translates a diagram written in the notation into conceptual graph form; it is called a *compilation algorithm*. A reverse translation algorithm for each notation — called an *extraction algorithm* — re-creates an original diagram from its conceptual graph representation. (These terms are taken from [19]). This paper summarizes several translation schemes, and provides an example for each of four notations. The compilation algorithms are summarized here; both the compilation and extraction algorithms are described in an Appendix to this paper. Further details are found in Appendix B of [20].

The next four sections of this paper each have the same structure: first, a notation is introduced, then its compilation algorithm summarized, followed by an example application of the translation. The particular notations in this paper were chosen because of their diversity, and because they are representative of the kinds of information typically captured by different requirements methods. The four notations are: entity-relationship (ER) diagrams, data flow diagrams, (DFDs) state transition diagrams (STDs), and requirements networks (R-nets) from SREM [2]. We make no claim that these are the "best" requirements notations, only that they are notations typically used by current software requirements developers.

The example system chosen is a patient-monitoring system for a hospital. This example in various forms has been treated by several previous authors on requirements (e.g., [2], [3], [21], [11], [7], [22]) as a reasonable system to illustrate requirements methods, since it exhibits several features of typical large complex software systems, e.g., real-time response, database access, potential for concurrency, etc.

3. ENTITY-RELATIONSHIP DIAGRAMS

This section illustrates the process of obtaining an R-Spec-Graph from an entity-relationship diagram. An entity-relationship (ER) diagram is a bipartite, non-directed graph where every node is either an entity or relation [23]. Nodes of either type may have associated attributes. Labels on arcs indicate the cardinality (many-to-many-ness) of a relation, e.g., one-to-one, one-to-many, etc. Entities in ER are denoted by a rectangle □, and relations in *R* are denoted by a diamond ◇. There are zero or more attributes (each denoted by a text string □ attribute) associated with an entity or relation. The graph's edges each have a many-to-many-ness label (default 1) in order to express one-to-one, one-to-many, or many-to-many relations. For example, 

means that r is a one-to-many (1 to M) relation — there is one A for several B's.

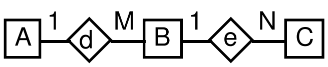
Since conceptual graphs are so close in structure to entity-relationships diagrams, our transformation is straightforward. We adopt these rules for translating its structure:

- 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].

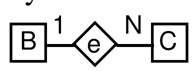
• An ER relation *r* becomes a conceptual graph relation (r).

Additional rules determine the cardinality of each referent, based on the cardinality of the original ER relation. Given the ER diagram in Figure 3(a), we initially transform it into Figure 3(b).

It is clear that entity A has cardinality 1, and entity C will have cardinality *N*, but what about entity B? Looking from left to right, B would have cardinality *N*, but from right to left, it would have cardinality 1. In this case, we select the entity C related to B whose many-to-many-ness is not one, then enclose B, C, and all its relations within a set referent in a context box. The context box then becomes the “entity” for further translations. In the example



, we translate the



portion into the conceptual graph of Figure 3(c) so the fragment becomes the graph in Figure 3(d). We may omit the cardinality indication (“@M”) when it means some indeterminate cardinality as in [T: { * }]. This point is discussed further at the end of Algorithm 1a in the Appendix.

Example Entity-Relationship Diagram

Consider the entity-relation diagram in Figure 4. This diagram was produced by the customer, who is responsible for the overall patient-monitoring system.

Transforming Figure 4 results in the specification graph shown in Figure 5. Note that arcs linked to the relations connection, plug_into, and alert are assigned a direction by the analyst, whereas the relation attribute has a pre-defined direction.

Discussion

The cardinality of the entities' referents are derived from the cardinality implied by the original ER relations. For example, the one-to-many ER relation plug_into results in the individual DATABASE having a cardinality of 1, whereas the individual MONITOR implies a cardinality of N for its enclosing context.

When translating the ER specification into its specification graph, either existing canonical graphs will determine the direction or with a completely new relation, a human analyst must decide which direction is appropriate. The direction of a newly-created relation's arrow is thus arbitrary, unless there are already-existing definitions (canonical graphs) specifying its direction. For an ER relation that is not in the canon, the direction of the arrow will reflect the dependency underlying the relationship. We therefore acknowledge that for some representations in this particular notation, the internal representation is less abstract than the original one. One can argue that any analyst-supplied directions ought to be subsequently added to the canon for later use.

4. DATA FLOW DIAGRAMS

This section illustrates the process of obtaining an R-Spec-Graph from a data flow diagram. A data flow diagram (DFD) consists of bubbles representing processes connected by labeled arrows denoting the flow of data between the bubbles. In order to represent data flow diagrams using conceptual graphs, we use actors, as described in [14]. An *actor* represents a data flow process that is allowed to change the referents of its outputs, using steps known as its private algorithm. Its arcs are classified as either input or output arcs, shown by dashed lines.

Our 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.

Example Data Flow Diagram

Consider the data flow diagram in Figure 6(a). This diagram was produced by a data base designer, who is responsible for determining the data base requirements of the patient monitoring system. Transforming Figure 6(a) results in the specification graph of Figure 6(b):

Discussion

The originating specification shows some fundamental limitations of data flow semantics, e.g., the Doctor_Nurse appears as a process that consumes an alarm and produces normal ranges, a somewhat different interpretation than our common sense would lead us to make. In future work, we want to show how other specifications can actually augment this one so that the role of the Doctor_Nurse is portrayed more accurately.

The effectiveness of translating data flow diagrams is made possible by the semantics of a conceptual graph actor, which models a process. Since actors occur in translating other notations (e.g., R-nets in §6), it would seem that the notion of a process is fundamental to modeling cognitive information. Without actors, many desired properties of real-world things could not otherwise be expressed using conceptual graphs. Creating an an actor via translation must be followed eventually by supplying an additional private algorithm describing how the actor alters its output concepts based on its input concepts.

5. STATE TRANSITION DIAGRAMS

This section illustrates the process of obtaining an R-Spec-Graph from a state transition diagram. We take the definition of state transition diagrams (STD) based on input tokens as described in [24]. We will translate state transition diagrams into conceptual graphs by using instances of the concept STATE to represent each state, a demon to represent each transition, and instances of DATA to represent each input and output token. The demon's semantics are described in [25]; they are somewhat similar to a Petri net place [26].

Since a demon denotes the creation of new concepts, and the retraction of existing ones, this representation of a state transition diagram is made simple. Only one state exists at any one time, because enabling the demon << transition >> causes the old state (and the input) to be retracted while it creates a new state (and the output). No explicit indicator of the current state is necessary; since only one state exists, it must be the current state. The start state is enabled by an *initiator demon* << T >>, thus creating the start state before any transitions occur. Our translation 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 token *J* becomes a conceptual graph concept [J].

J < DATA 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 event *J* to state *T* with output event *K* becomes a demon with links to [T] and [K], and with links from [S] and [J]. K < DATA and T < STATE are included in the type hierarchy.

Example State Transition Diagram

Consider the state transition diagram of Figure 7. This diagram was produced by an enduser (a doctor or nurse) who is interested in the actions of the patient-monitoring system during its operation. States are indicated by ovals; transitions are shown as input-event / output-event (with "—" denoting a null event). Transforming Figure 7 results in the specification graph of Figure 8.

Discussion

The effective translation of a state transition diagram is made possible by the semantics of the demon, which we introduced in [20] and discussed further in [25]. Since a state transition diagram expresses a set of dynamically changing STATEs, its true semantics cannot be captured by an actor. The demon's semantics are appropriate for state transition diagrams. The loopback arc from STATE nominal readings is represented by a << transition >> demon that re-creates its input state. Figure 8 effectively "eats" its input data tokens (whose origin is not specified), and over time produces a series of output tokens (whose disposition is not specified). The output tokens accumulate (remain in existence), since they do not serve as input to any demons.

If state transition diagrams were extended to accommodate multiple concurrent states, these algorithms can be adapted with only minor modifications, to represent multiple start states, and multiple concurrent transitions. There is no restriction on the number of demons active at any one time, so many transitions may take place simultaneously. Likewise, there is no restriction on the number of concepts that may exist at any time, so many different states (and their output tokens) may exist simultaneously. Thus conceptual graphs with demons are an adaptable representation for systems that change state.

A more accurate (and more complex) transformation would better capture the notion that the arrival of an input token is an event. This would allow us to model generalized state transition diagrams where a state transition is triggered by some arbitrary event instead of the particular event of an input token arriving. The diagram Figure 8 is somewhat suggestive in that regard, since certain data “tokens” (e.g., READINGS_OFF_NORMAL) suggest an implied event.

6. REQUIREMENTS NETWORKS (SREM)

This section illustrates the process of obtaining an R-Spec-Graph from a SREM R-net [2]. This translation scheme makes use of a component-for-component equivalence table in Figure 9 which shows R-net components and their corresponding structures in conceptual graphs.

This translation relies on properties of first-order logic that are explained in [14]. In particular, a logical OR relation is represented through an application of deMorgan’s law; i.e., $P \vee Q = \neg(\neg P \wedge \neg Q)$. This is because a logical AND relation is easier to express in conceptual graphs — the appearance of two graphs in a shared context is equivalent to their conjunction. Thus the graph ¬ [¬ [P] ¬ [Q]] denotes the logical relation $P \vee Q$. This is the origin of the several negated contexts in Figure 9 and Figure 10(b).

The selector ⊕ needs some explanation. It must represent knowledge about several alternatives, even though the selector chooses only one. Translating a selector results in several conceptual graph contexts, one for each alternative. The alternatives are made mutually exclusive by placing the selector in a generic concept in the current context (e.g., [ITEM]), while placing individual selector instances (e.g., [ITEM: a], [ITEM: b], etc.) in new separate contexts. Each individual instance of a selector value is connected to the generic concept (e.g., [ITEM: {a, b, else} @1]) by a line of identity, but is not connected to the other selector individuals; e.g., [ITEM: a] is not connected by a line of identity to [ITEM: b]; they are not compatible. Because the cardinality of [ITEM] is 1, however, only one of the individuals (values) can exist at any one time.

Example Requirements Network

Consider the requirements network (R-net) diagram in Figure 10(a) taken from [2]. This diagram is produced by a software tester, who is responsible for determining that certain functions of the delivered patient-monitoring system meet its stated requirements. Transforming Figure 10(a) results in the specification graph of Figure 10(b).

Discussion

The nested contexts in Figure 10(b) may interfere with understanding, particularly for a large R-net. Some shorthand notations have been proposed for conceptual graphs that provide new constructs to express some first-order logic conditions more naturally. Our main purpose here is to show that all the information in an R-net can be captured by conceptual graphs.

This transformed R-net's graph does not explicitly capture the real-time nature of the stimulus/response paradigm. One goal of future work is to show how explicit underlying assumptions can augment the knowledge represented by a participant's R-Spec-Graph.

7. SUMMARY

We showed how to translate several different notations into conceptual graphs. The multiple-viewed approach accommodates existing notations and therefore is compatible with some existing software requirements development procedures. Using conceptual graphs can be useful in understanding even a single specification; however, it is their generality which will be most important when we go about analyzing multiple views.

Extraction — transforming conceptual graphs back into one of the originating notations — can be useful for two purposes: (1) to show which conceptual graph structures capture the notation's information (e.g., only concepts and relations are present in an ER-based conceptual graph), and (2) to express new graphs after we have identified overlapping conceptual graphs and want to convey our results to participants in their originating notations.

With translation schemes available, a requirements analyst can use several different languages to represent different aspects of a large system's requriements, or several different analysts can each use a different language of their choice. The representations can then be represented internally using the single language of conceptual graphs. Using methods that are not described here, the representations can be compared and combined automatically. The results can then be extracted from the internal representation and translated back to the several different analyst's languages to be examined and evaluated. Further information may then be supplied, followed by further automatic analysis.

These results have implications beyond an aid to representing the knowledge in software requirements. Several notations are used in other phases of software development, notably design. Wherever these notations are useful, their translation into and from conceptual graphs can be augmented by knowledge-based storage and analysis methods.

Once requirements specifications in different notations are translated into conceptual graphs, we have a basis for comparing them. Conceptual graphs are a promising avenue for exploring formally the interaction between multiple requirements views. In future work we intend to describe how to determine the overlap between these multiple specifications.

APPENDIX: TRANSLATION SCHEMES

This appendix gives detailed algorithms showing how we translate requirements specifications into conceptual graphs. The two algorithms that translate a diagram into conceptual graphs and back are called a *translation scheme*. One algorithm translates the notation into conceptual graph form; it is called a *compilation algorithm* (the "a" algorithms below). To complete the descriptions, we also provide a reverse translation algorithm for each notation that re-creates an original diagram from its conceptual graph representation; it is called an *extraction algorithm* (the "b" algorithms below). These terms are taken from Muehlbacher's work [19].

Each section of this appendix is organized as follows: first, the notation is described, then its translation algorithms are shown. An originating specification in the notation is referred to as [_ **LANG** X R-Spec _], its resulting conceptual graph representation is referred to as [**LANG** X R-Spec-Graph].

In the following translation algorithms, whenever a concept is added to a graph, the algorithm will specify that the added concept will be joined with a line of identity to any previous occurrence of that concept.

For each notation, we assume we are starting from a well-formed R-Spec diagram. This is for two reasons: first, because it simplifies the algorithms and second, these notations already have well-established techniques for determining their internal consistency. We therefore do not check for badly-formed originating diagrams.

Entity-Relationship Translation Scheme

An entity-relationship diagram is a bipartite, non-directed graph where every node is either an entity or relation [23]. Nodes of either type may have associated attributes. Labels on arcs indicate the arity (many-to-many-ness) of a relation, e.g., one-to-one, one-to-many, etc. Our general strategy is to translate entities into conceptual graph concept boxes and relations into conceptual graph relations, where the relation's dependency is either obtained from pre-existing relation definitions or supplied by the analyst.

Algorithm 1a provides a translation from [_ **LANG** era R-Spec _] to [**LANG** era R-Spec-Graph: *U*]. **Algorithm 1b** provides a translation back to [_ **LANG** era R-Spec _] from [**LANG** era R-Spec-Graph: *U*]. Therefore all information in [_ **LANG** era R-Spec _] must be contained in [**LANG** era R-Spec-Graph: *U*].

Compilation Algorithm 1a

A compilation algorithm for translating the entity-relationship diagram [_ **LANG** era R-Spec _] into [**LANG** era R-Spec-Graph: *U*] is the following:

Let [_ **LANG** era R-Spec _] be a non-directed bipartite graph whose vertices are a set of entities *C*, and a set of relations *R*. There are zero or more attributes associated with each vertex. The graph's edges consist of the set *E*, each member having a many-to-many-ness label (default 1) in order to express one-to-one, one-to-many, or many-to-many relations.

Begin Algorithm 1a.
Let $u = \{ \}$.
Each node and edge in [_ **LANG** era R-Spec _] is initially unmarked.

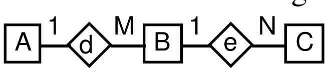
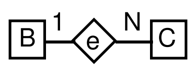
While $\exists r \in R$ such that *r* is not marked **loop**
    Put "(r)" in *u.* ; *translates relations*
    Mark *r.*
    Look for canonical definition of *r* (if any) to determine direction of links.
    If there are any attributes associated with *r* **then begin**
        Link "→ [Attrset: *r*]" to (r) in *u.*
        ; *kludge to account for attributes of relations since*
        ; *can't link 2 relations:* (rel) → (attribute) → [T]
        For each attribute a_i such that a_i is associated with *r* **loop**
            Link "→ (attribute) → [a_i]" to [Attrset: *r*] in *u* ; *translates relation's attributes*
        end for
    end if

```
    For each edge e_i connected to r loop
        Let c_i be the entity to which it is connected.
        Let m_i be its many-to-many-ness label.
        If ∃c_k ∈ C such that:
                c_k ≠ c_i and c_k is connected to r and c_k has cardinality m_k ≠ 1 then
                                    ; direction of arrows below (dependency) is either:
                                    ;   1. obtained from canonical definition of r, if present.
                                    ;   2. obtained through manual means.
            Put "[ { [ c_k : { * } ] ← (r) ← [ c_i : { * } @m_i ] }  @m_k ]" around (r) in u.
                                                            ; (r) is already there
                                                            ; see discussion in text below.
            If c_k is already marked then
                Join this occurrence of [c_k ] to previous occurrence with same referent (see text).
            else begin
                Mark c_k
                Put "c_k < ENTITY." into u.
                For each attribute a_kj such that a_kj is associated with c_k loop
                    Link "→ (attribute) → [a_kj ]" to c_k in u    ; translate related entity's attributes
                    Put "a_kj < ATTRIBUTE." into u.
                end for
            end if
        else
            Link "← [ c_i : { * } @m_i ]" to (r) in u                 ; translate related entities
        end if
                    Special case: if all m_i = 1, then link "← [c_i ]" to (r) in u
                        because [ T : { * } @1] is equivalent to [ T: * ]
        If c_i is already marked then
            Join this occurrence of [c_i ] to a previous occurrence with same referent (see text).
        else begin
            Mark c_i
            Put "c_i < ENTITY." into u.
            For each attribute a_ij such that a_ij is associated with c_i loop
                Link "→ (attribute) → [a_ij ]" to c_i in u          ; translate related entity's attributes
                Put "a_ij < ATTRIBUTE." into u.
            end for
        end if
    end for
end while

While ∃c ∈ C such that c is not marked loop
    Put "[ c ]" in u.                                          ; translate any unrelated entity
    Mark c.
    Put "c < ENTITY." into u.
    For each attribute a_i such that a_i is associated with c loop
        Link "→ (attribute) → [a_i ]" to c in u                  ; translate its attributes
        Put "a_i < ATTRIBUTE." into u.
    end loop
end while

end Algorithm 1a.
```

Although this algorithm will preserve a relation's attributes, we have not yet developed any techniques for manipulating them further; our analysis will be based on entities' attributes only.

The handling of many-to-many-ness in Algorithm 1a was introduced in §3 of the paper. Consider the ER diagram in Figure 3(a). Using Algorithm 1a as given above, we obtain the graph in Figure 3(b). It is clear that concept A is a set of cardinality 1, and concept C is a set of cardinality *N*, but what about concept B? Looking from left to right, B should become a set of cardinality *N*, but from right to left, it should become a set of cardinality 1. When this situation occurs, the solution is to select the entity E related to B whose many-to-many-ness is not one, then enclose B, E, and all its relations within a set referent in a context box. The entire context box then becomes a single "entity" for further translations. In the example , we translate the  portion into the graph of Figure 3(c), so the example diagram is translated into Figure 3(d).

We may omit the cardinality indicator ("@M") when it represents an indeterminate cardinality as in [T: { * }].

Algorithm 1b shows how the original ER R-Spec can be obtained from its corresponding conceptual graph representation. The reader may wish to look at the example in §3 in the paper before examining **Algorithm 1b**.

Extraction Algorithm 1b

The following extraction algorithm translates [**LANG** era R-Spec-Graph: *u*] to [_ **LANG** era R-Spec _]. The graph [**LANG** era R-Spec-Graph] consists of three finite sets: *C*, the set of all its concepts; *R*, the set of all its relations; and *E*, the set of its directed links. A link contains the identity of its two ends. Any actors or demons in [**LANG** era R-Spec-Graph] are ignored.

```
Begin Algorithm 1b.
Let R-Spec be blank.
Each concept, relation and link in u is initially unmarked.

While ∃c ∈ C such that c is not marked loop
   If c has no incoming link from relation (attribute) then begin
      Put "entity c" in R-Spec.                                   ; entities preserved.
      Mark concept c.
      For each c_i ∈ C such that "c → (attribute) → c_i" is in u loop
         Put "attribute c_i" with entity c in R-Spec.     ; entity's attributes preserved.
         Mark c_i .
         Mark (attribute).
         Mark both links to (attribute) relation.
      end loop
   end if
end loop

For each r_i ∈ R such that r_i is not (attribute) loop       ; attributes already preserved.
   Put "relation r_i" into R-Spec.                               ; relations preserved.
   Mark r_i.
   for each c ∈ C such that link "c → ( r_i )" or "c ← ( r_i )" is in u loop
      if c = [ Attrset ] then begin
         for each a_i ∈ C such that "c → (attribute) → a_i" is in u loop
            Put "attribute a_i" with entity c in R-Spec.  ; relation's attributes preserved.
            Mark a_i.                                      ; (see kludge in Algorithm 1a).
            Mark relation (attribute).
            Mark both links to (attribute) relation.
         end loop
      else begin                              ; normal connections from relation to entity.
         Connect era relation r_i to entity c in R-Spec.
         if c ∈ C is within a nested context in u then begin
            Let k be the cardinality of the context.        ; many-to-many-ness preserved.
            Put "many-to-many-ness = k" with link r_i to c in R-Spec.
         else
            Put "many-to-many-ness = 1" with link r_i to c in R-Spec.
         end if
      end if
   end loop
end loop

end Algorithm 1b.
```

This concludes the translation scheme for entity-relationship diagrams.

Data Flow Translation Scheme

In this section, we use data flow diagrams based on the Yourdon/de Marco variant [27], [28]. In order to represent data flow diagrams using conceptual graphs, we use actors, as explained in §4 above. An *actor* represents a data flow process that is allowed to change the referents of its outputs, using steps known as its *private algorithm*. Its arcs are classified as either *input* or *output arcs*, shown by dashed lines.

Algorithm 2a provides a translation from [_ **LANG** dfd R-Spec _] to [**LANG** dfd R-Spec-Graph: *U*]. **Algorithm 2b** provides a translation back to [_ **LANG** dfd R-Spec _] from [**LANG** dfd R-Spec-Graph: *U*]. Therefore all information in [_ **LANG** dfd R-Spec _] must be contained in [**LANG** dfd R-Spec-Graph: *U*].

Compilation Algorithm 2a

A compilation algorithm for translating the data flow diagram [_ **LANG** dfd R-Spec _] into [**LANG** dfd R-Spec-Graph: *U*] is the following:

Let the private referent of [_ **LANG** dfd R-Spec _] be a directed connected graph consisting of a set of labeled bubbles P representing processes, and a set of labeled directed edges E representing data flows. An edge connected to only one bubble represents either a data source or data sink; it is handled by the algorithm as an incomplete arc. Every edge is connected to at least one bubble .

The following algorithm translates [_ **LANG** dfd R-Spec _] into [**LANG** dfd R-Spec-Graph:*U*].

```
Begin Algorithm 2a.
Let u = { }. Each bubble and edge in SPEC is initially unmarked.

While ∃p ∈ P such that p is not marked loop
   Put "< p >" in u                                          ; actors preserve bubbles
   For all e_i such that e_i ∈ E and e_i is connected to p loop
      If e_i is a plural noun then
         Let e = singular form of e_i.
         Let c = [ e: { * } ]
      else
         Let c = [ e_i ]
      end if
      If e_i is directed towards p then link "← c" to < p > in u
      else link "→ c " to < p > in u                          ; concepts preserves data
      end if
      if e_i is marked then
         Join this occurrence of [e_i ] to a previous occurrence with same referent (see text).
      else
         Mark e_i
         Put "e_i < DATA." into u.
      end if
   end loop
end loop

end Algorithm 2a.
```

Since in a well-formed data flow diagram, every edge must be connected to at least one process, this algorithm terminates when every $p \in P$ has been marked, and therefore every edge has been translated.

Algorithm 2b below shows how the original DFD R-Spec can be obtained from its corresponding conceptual graph representation. The reader may wish to look at the example of §4 in the paper before examining **Algorithm 2b**.

Extraction Algorithm 2b

The following extraction algorithm translates [**LANG** dfd R-Spec-Graph] into [_ **LANG** dfd R-Spec _]. The graph [**LANG** dfd R-Spec-Graph: *u*] consists of three sets: *A*, the set of its actors; *R*, the set of its relations; and *E*, the set of its directed links. A link includes the identity of its two ends. Any relations or demons in [**LANG** dfd R-Spec-Graph] are ignored. Every concept is connected by either one link or two links to an actor; if connected by two links, one must be an incoming link, the other an outgoing link.

```
Begin Algorithm 2b.
Let R-Spec be blank.
Each concept and relation in u is initially unmarked.

for each a ∈ A loop
   Put "process a" in R-Spec.                          ; bubbles preserved.
   Mark actor a
end loop
for each c ∈ C loop
   if c has one link e_i toward actor a_i then begin       ; identify sources.
      Put "source c" on edge directed towards process a_i in R-Spec.
      Mark c.                                          ; preserve sources.
   end if
   if c has one link e_k away from actor a_k then begin      ; identify sinks.
      Put "sink c" on edge directed from process a_k in R-Spec.
      Mark c.                                          ; preserve sinks.
   end if
   if c has one link e_i toward actor a_i and
        one link e_k toward actor a_k then begin   ; identify intermediate data.
      Put "c" on edge directed from process a_k towards process a_i in R-Spec.
      Mark c.                               ; preserve intermediate flows.
   end if
end loop

end Algorithm 2b.
```

This concludes the translation scheme for data flow diagrams.

State Transition Translation Scheme

We use the standard definition of state transitions based on input symbols [24], although we may be able to generalize our definition to include any event as an input symbol or output symbol. In the standard definition, each event therefore means the arrival of an input symbol or the creation of an output symbol. We will translate state transition diagrams into conceptual graphs by using instances of the concept STATE to represent each state, a demon to represent each transition, and instances of DATA to represent input and output tokens.

Algorithm 3a provides a translation from [_ **LANG** std R-Spec _] into [**LANG** std R-Spec-Graph: *u*]. **Algorithm 3b** provides a translation back to [_ **LANG** std R-Spec _] from [**LANG** std R-Spec-Graph: *u*]. Therefore all information in [_ **LANG** std R-Spec _] must be contained in [**LANG** std R-Spec-Graph: *u*].

Let the state transition diagram be denoted by a private referent of the concept [_ **LANG** std R-Spec: $(Q, \Sigma, \Delta, \delta, q_0, F)$ _], where the elements of the 6-tuple are:

Q = finite set of states,

Σ = finite set of input tokens,

Δ = finite set of output tokens,

δ = map from $Q \times \Sigma \rightarrow (Q, \Delta)$ the state transition function,

$q0 \in Q$ the start state, and

$F \supset Q$ the set of final (accepting) states.

The state transition semantics associated with the definition assumes the following:

• Every time an input token *s* appears, the state transition function δ is examined to find some $Q \times \Sigma$ pair $(q_{current}, s)$ whose result is (q_{next}, d). The state q_{next} becomes the current state, and output token *d* is produced.

• Before any action occurs, state q_0 is the current state.

Compilation Algorithm 3a

A compilation algorithm for translating the state transition diagram [_ **LANG** std R-Spec _] into [**LANG** std R-Spec-Graph: U] is the following:

```
Begin Algorithm 3a.
Let u = { }. All elements of all sets (Q, Σ, Δ, δ, q0, and F) are initially unmarked.

While ∃q ∈ Q such that q is not marked begin
   Mark q
   Put "q < STATE." into u.
   If q = q0 then
      Put "<<T >> → [ q ]" into u.                    ; preserves initial states
   else
      Put "[ q ]" into u.                              ; preserves other states
   end if
end while

While ∃s ∈ Σ such that s is not marked begin
   Put "[ s ]" into u.          ; preserve input tokens
   Put "s < DATA." into u.
   Mark s.
end while

While ∃d ∈ Δ such that d is not marked begin
   Put "[ d ]" into u.                                 ; preserve output tokens
   Put "d < DATA." into u.
   Mark d.
end while

While ∃f ∈ δ such that f is not marked begin
   Let qi ∈ Q be f 's current state.
   Let s ∈ Σ be f 's input token.
   Let qj ∈ Q be f 's next state.
   Let d ∈ Δ be f 's output token.
   Put "[ qi ] → <<transition >>—
         ← [ s ],
         → [ qj ],
         → [ d ]"  into u.                             ; preserves transition functions
   Join this occurrence of [ qi ] to a previous occurrence with same referent.
   Join this occurrence of [ s ] to a previous occurrence with same referent.
   Join this occurrence of [ qj ] to a previous occurrence with same referent.
   Join this occurrence of [ d ] to a previous occurrence with same referent.
   Mark f.
end while
While ∃f ∈ F such that f is not marked begin
   Put "[ f ] ← (final)" into u                        ; preserves final states.
      Join this occurrence of [ f ] to a previous occurrence with same referent.
end while

end Algorithm 3a.
```

Since a demon denotes the creation of new concepts, and the retraction of existing ones, this representation of a state transition diagram is made simple. Only one state exists at any one time, because enabling the demon << transition >> causes the old state (and the input) to be retracted while it creates a new state (and the output). No explicit indicator of the current state is necessary; since only one state exists, it must be the current state. The start state is enabled by demon <<T >> (the initiator demon, see Appendix A, [20]), thus causing the start state to exist before any transitions occur.

Algorithm 3b below shows how the original state transition R-Spec can be obtained from its corresponding conceptual graph representation. The reader may wish to look at the example in §5 of the paper before examining **Algorithm 3b**.

Extraction Algorithm 3b

The following extraction algorithm translates [**LANG** std R-Spec-Graph: *u*] to [_ **LANG** std R-Spec _]. The graph [**LANG** std R-Spec-Graph: *u*] consists of four finite sets: *C*, the set of its concepts; *R*, the set of its relations; *D*, the set of its demons; and *E*, the set of its directed links. Only relations named (final) are in *R*, and *D* contains only demons named <<transition >> or << T >> (the initiator). Any other demons or relations, and all actors in *u* are ignored.

Begin Algorithm 3b.
Let R-Spec contain six sets Q, Σ, Δ, δ, q_0, and F.
where $Q = \{\ \}$, $\Sigma = \{\ \}$, $\Delta = \{\ \}$, $\delta = \{\ \}$, $q_0 = \{\ \}$, $F = \{\ \}$.

For each *s* such that "<<T >> → [*s*]" and "*s* < STATE." are in *u* **loop**
Let $q_0 = q_0 \vee \{ s \}$ *; preserves start state.*
end for

For each $[s] \in C$ such that "<<T >> → [*s*]" and "*s* < STATE." are in *u* **loop**
Let $Q = Q \vee \{ s \}$ *; preserves set of all states.*
end for

For each $d \in D$ such that d = “<<transition >>” and
“[q] → d → [s]” and “q < STATE.” and “s < STATE.” are in u **loop**
If “[inp] → d” and “inp < DATA.” are in u **then begin**
Let $i = inp$
Let $\Sigma = \Sigma \vee \{ inp \}$ *; preserves set of input tokens.*
else
Let i = “—” *; in case of no input token.*
end if
If “d → [out]” and “out < DATA.” are in u **then begin**
Let $o = out$
Let $\Delta = \Delta \vee \{ out \}$ *; preserves set of output tokens.*
else
Let o = “—” *; in case of no output token*
end if
Let $\delta = \delta \vee \{ f \}$ where *; preserves transition functions.*
q is f's current state,
i is f's input token,
r is f's next state, and
o is f's output token.
end for
For each [s] ∈ C such that “(final) → [s]” is in u **loop**
Let $F = F \vee \{ s \}$ *; preserves set of final states.*
end for

end Algorithm 3b.

This concludes the translation scheme for state transition diagrams.

Requirements Network Translation Scheme

This translation scheme makes use of a component-for-component equivalence table which shows the major R-net components and their corresponding structures in conceptual graphs. The same table will be used for both algorithms.

Algorithm 4a provides a translation from [_ **LANG** Rnet R-Spec _] to [**LANG** Rnet R-Spec-Graph: U]. **Algorithm 4b** provides a reverse translation back to [_ **LANG** Rnet R-Spec _] from [**LANG** Rnet R-Spec-Graph: U]. Therefore all information in [_ **LANG** Rnet R-Spec _] must be contained in [**LANG** Rnet R-Spec-Graph: U].

Compilation Algorithm 4a

A compilation algorithm for translating a requirements network diagram [_ **LANG** Rnet R-Spec _] into [**LANG** Rnet R-Spec-Graph: U] is the following:

Begin Algorithm 4a.
Let *u* = { }. All elements of [_ **LANG** Rnet R-Spec _] are initially unmarked.
For each *s* such that *s* is a start symbol **loop**
  For each node *n* in the path starting with *s* **loop**
    Find node *n* in the **Symbol** column of Figure 9.
   Re-write *n* in conceptual graph form as **In Conceptual Graphs** in Figure 9.
    Mark *n*.
  end loop
end loop

End of Algorithm 4a.

The table in Figure 9 lists the component-for-component translations which are implemented by Algorithms 4a and 4b.

This algorithm relies on some properties of first-order logic which are explained in §2 and 14. The selector (⨁) is also explained in §2 as follows. It must represent knowledge about several alternatives, even though the selector chooses only one. Translating a selector results in several conceptual graph contexts, one for each alternative. The alternatives are made mutually exclusive by placing the selector in a generic concept in the current context (e.g., [ITEM]), while placing individual selector instances (e.g., [ITEM: a], [ITEM: b], etc.) in new separate contexts. Each individual instance of a selector value is connected to the generic concept (e.g., [ITEM: {a, b, else} @1]) by a line of identity, but is not connected to the other selector individuals; e.g., [ITEM: a] is not connected by a line of identity to [ITEM: b]; they are not compatible. Because the cardinality of [ITEM] is 1, however, only one of the individuals (values) can exist at any one time.

Algorithm 4b below shows how the original requirements network R-Spec can be obtained from its corresponding conceptual graph representation. The reader may wish to look at the example in §6 of the paper before examining **Algorithm 4b**.

Extraction Algorithm 4b

The following extraction algorithm translates [**LANG** Rnet R-Spec-Graph: *u*] to [_ **LANG** Rnet R-Spec _]. The graph [**LANG** Rnet R-Spec-Graph: *u*] consists of four finite sets: *C*, the set of its concepts; *R*, the set of its relations; *D*, the set of its demons; and *E*, the set of its directed links.

Begin Algorithm 4b.

Let R-Spec be blank.

For each context *x* in *u*, starting with outermost context **loop**
  For each component in *x*, **loop**
    Find a match for *x* in the **In Conceptual Graphs** column of Figure 9.
    Get an R-net component *c* from the **Symbol** column of Figure 9
    Connect *c* to R-Spec.
  end for
end for

End of Algorithm 4b.

This concludes the translation scheme for requirements networks (RSL).

REFERENCES

[1] G.C. Roman, A Taxonomy of Current Issues in Requirements Engineering, *IEEE Computer* 18 (4), 14-23 (1985).

[2] M.W. Alford, A Requirements Engineering Methodology for Real-Time Processing Requirements, *IEEE Trans. Softw. Eng.* SE-3 (1), 60-69 (1977).

[3] M.W. Alford, SREM at the Age of Eight: The Distributed Computing Design System, *IEEE Computer* 18 (4), 36-46 (1985).

[4] D.T. Ross and J. Kenneth E. Schoman, Structured Analysis for Requirements Definition, *IEEE Trans. Softw. Eng.* SE-3 (1), 6-15 (1977).

[5] D.T. Ross, Applications and Extensions of SADT, *IEEE Computer* 18 (4), 25-35 (1985).

[6] G.P. Mullery, CORE - A Method For Controlled Requirement Specification, *Proc. 4th Intl. Conf. Softw. Eng.*, p. 126-135, IEEE Computer Society 1979.

[7] A. Borgida, S. Greenspan, and J. Mylopoulos, Knowledge Representation as the Basis for Requirements Specifications, *IEEE Computer* 18 (4), 82-91 (1985).

[8] S.C. Bailin, An Object-Oriented Requirements Specification Method, *Comm. ACM* 32 (5), 608-623 (1989).

[9] D. Harel, On Visual Formalisms, *Comm. ACM* 31 (5), 514-530 (1988).

[10] A.I. Wasserman, P.A. Pircher, D.T. Shewmake, and M.L. Kersten, Developing Interactive Information Systems With the User Software Engineering Methodology, *IEEE Trans. Softw. Eng.* SE-12 (2) (1986).

[11] S.P. Wartik, A Multi-Level Approach to the Production of Requirements for Interactive Computer Systems, Ph.D. Thesis, University of California, Santa Barbara, 1983.

[12] W.W. Agresti, Guidelines for Applying the Composite Specification Model, Software Engineering Laboratory, Tech. Report no. SEL-87-003, Greenbelt, MD, U.S.A., 1987.

[13] A. Finkelstein and H. Fuks, Multiparty Specification, *Proc. 5th Intl. Wkshop, Softw. Spec. and Design*, p. 185-195, ACM SIGSOFT 1989.

[14] J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publ. Co., Reading, Mass. U.S.A. 1984.

[15] R.C. Schank, *Conceptual Information Processing*, Elsevier Science Publishers 1975.

[16] *Proc. Fourth Annual Workshop on Conceptual Structures*, J.A. Nagle and T.E. Nagle, eds., AAAI, IJCAI-89, Detroit, Michigan, 1989.

[17] *Proc. Fifth Annual Workshop on Conceptual Graphs*, P. Eklund and L. Gerholz, eds., Linköping University, Stockholm, Sweden, 1990.

[18] *Proc. Sixth Annual Workshop on Conceptual Graphs*, E.C. Way, ed., SUNY Binghamton, Binghamton, NY, 1991.

[19] R. Muehlbacher, Using Conceptual Graphs as a Representation Language for System Analysis Methods, *Proc. 5th Annual Workshop on Conceptual Structures*, (P. Eklund and L. Gerholz, eds.), p. 221-232, Linköping University, Boston & Stockholm 1990.

[20] H.S. Delugach, A Multiple-Viewed Approach to Software Requirements, Ph.D. Thesis, University of Virginia, Charlottesville, VA U.S.A., 1991.

[21] P. Zave, An Operational Approach to Requirements Specification for Embedded Systems, *IEEE Trans. Softw. Eng.* SE-8 (3), 250-269 (1982).

[22] A.M. Davis, *Software Requirements: Analysis and Specification*, Prentice-Hall, Englewood Cliffs, New Jersey 1990.

[23] P.P.S. Chen, The entity-relationship model — toward a unified view of data, *ACM Trans. Database Sys.* 1 (1), 9-36 (1976).

[24] A.V. Aho, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Vol. I, Englewood Cliffs, New Jersey 1972.

[25] H.S. Delugach, Dynamic Assertion and Retraction of Conceptual Graphs, *Proc. Sixth Annual Workshop on Conceptual Graphs*, (E.C. Way, ed.), p. 15-26, SUNY Binghamton, Binghamton, New York 1991.

[26] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey 1981.

[27] T. DeMarco, *Structured Analysis: Systems Specifications*, Yourdon, Prentice-Hall, New York 1980.

[28] E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, New Jersey 1979.

FIGURES

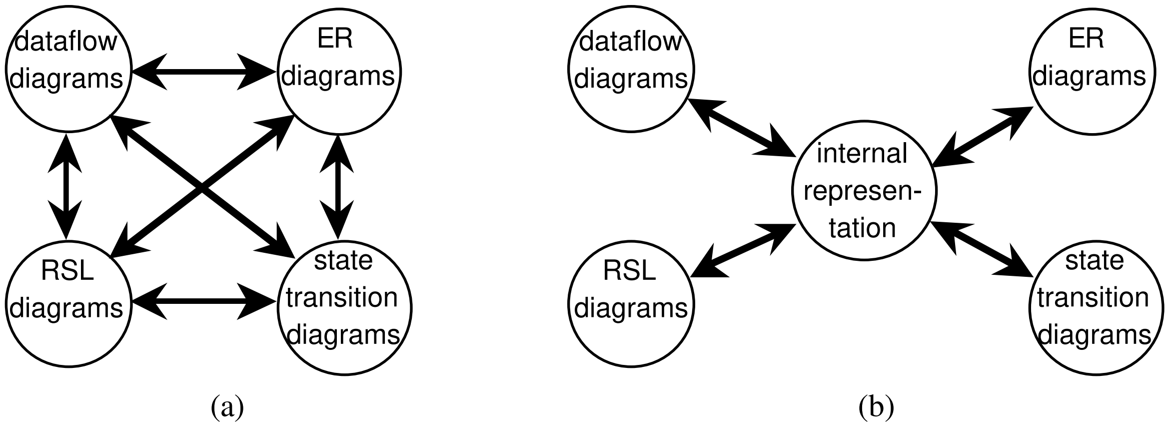


Figure 1. **Translating Between Requirements Languages.**

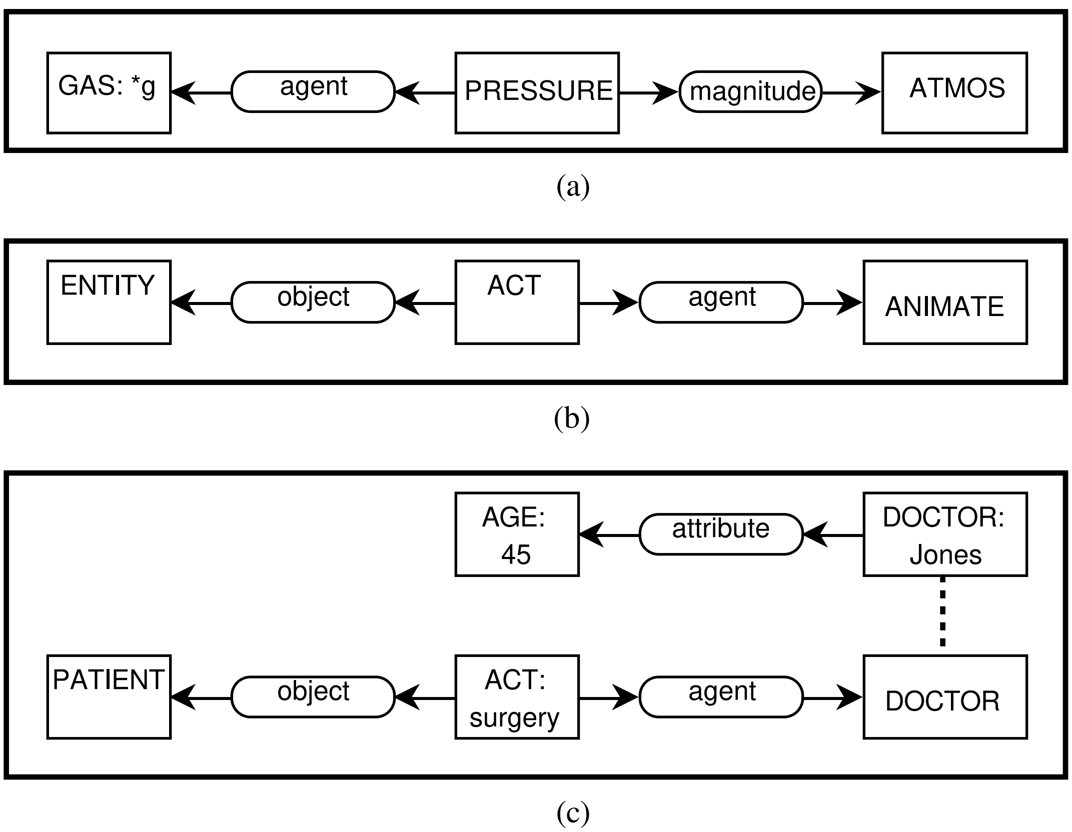


Figure 2. **Conceptual Graph Examples.**

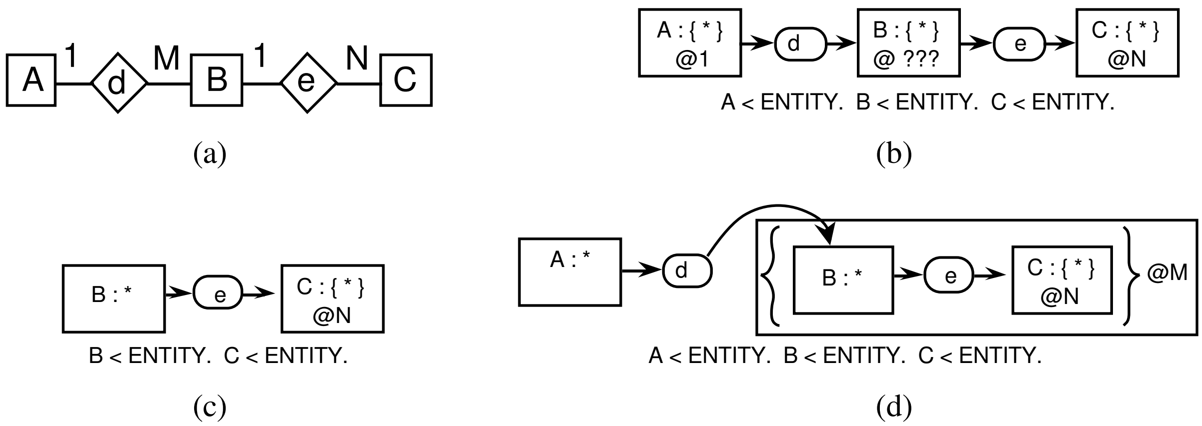


Figure 3. **Translating Cardinality From ER Diagrams.**

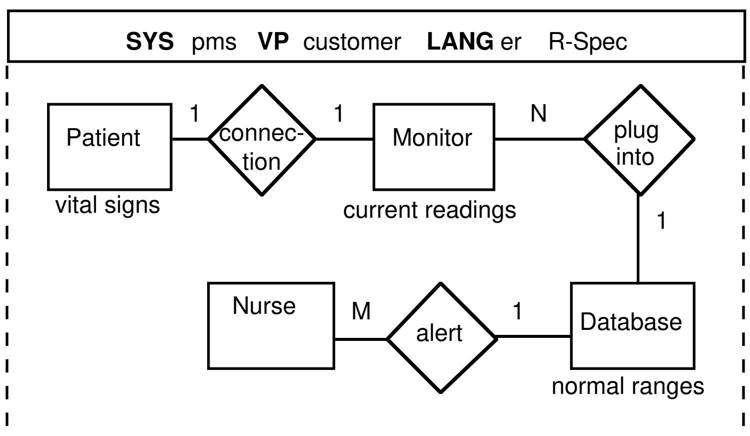
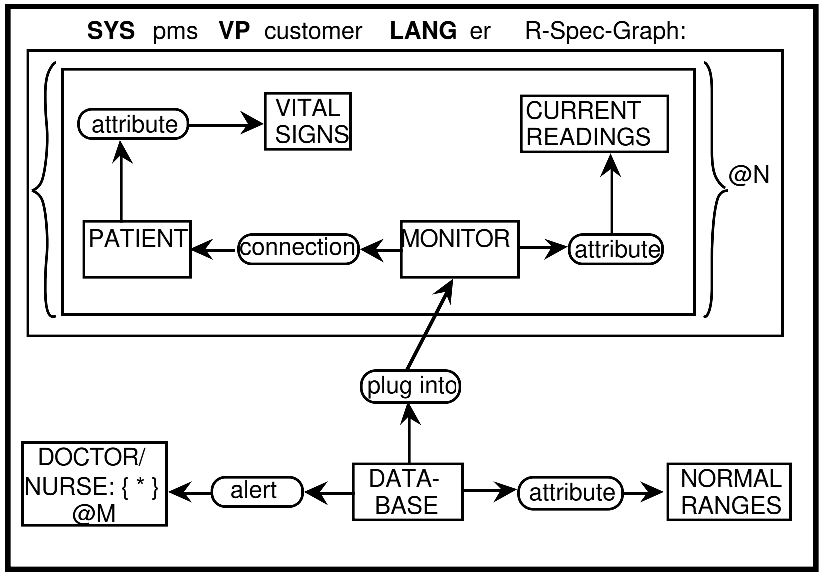
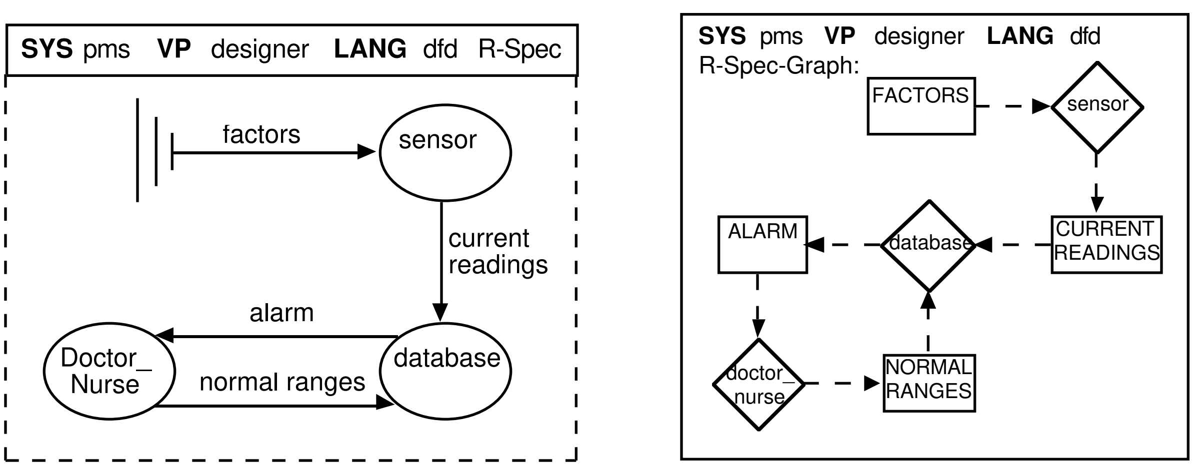


Figure 4. **Customer's Originating Entity-Relation Diagram.**



VITAL_SIGNS < ATTRIBUTE.
NORMAL_RANGES < ATTRIBUTE.
MONITOR < ENTITY.
DOCTOR_NURSE < ENTITY.
CURRENT_READINGS < ATTRIBUTE.
PATIENT < ENTITY.
DATABASE < ENTITY.

Figure 5. **Conceptual Graph From Customer's Entity-Relation Diagram.**



FACTORS < DATA. ALARM < DATA.
CURRENT_READINGS < DATA.
NORMAL_RANGES < DATA.

(a) (b)

Figure 6. **Designer's Originating and Transformed Data Flow Diagram.**

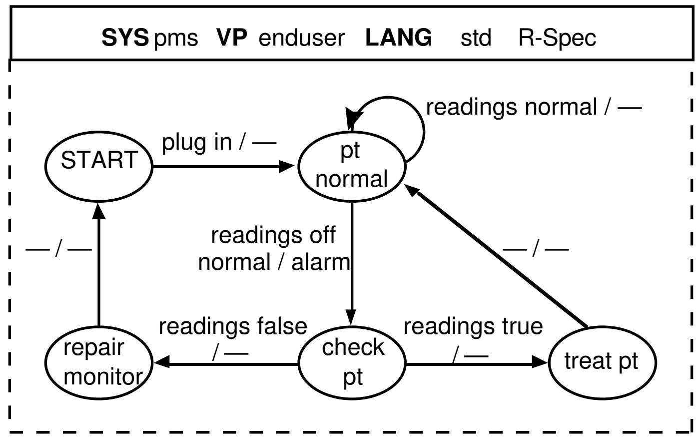
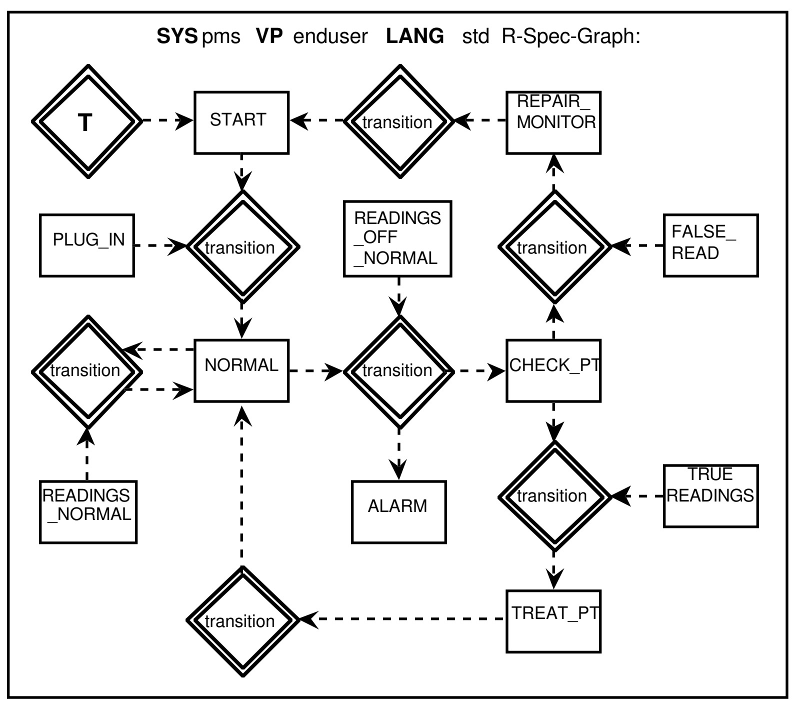


Figure 7. **End-User's Originating State Transition Diagram.**



START < STATE. NORMAL < STATE. CHECK_PT < STATE.
TREAT_PT < STATE. REPAIR_MONITOR < STATE. PLUG_IN < DATA.
READINGS_OFF_NORMAL < DATA. ALARM < DATA.
READINGS_NORMAL < DATA. TRUE_READINGS < DATA. FALSE_READ < DATA.

Figure 8. **Conceptual Graph From End-User's State Transition Diagram.**

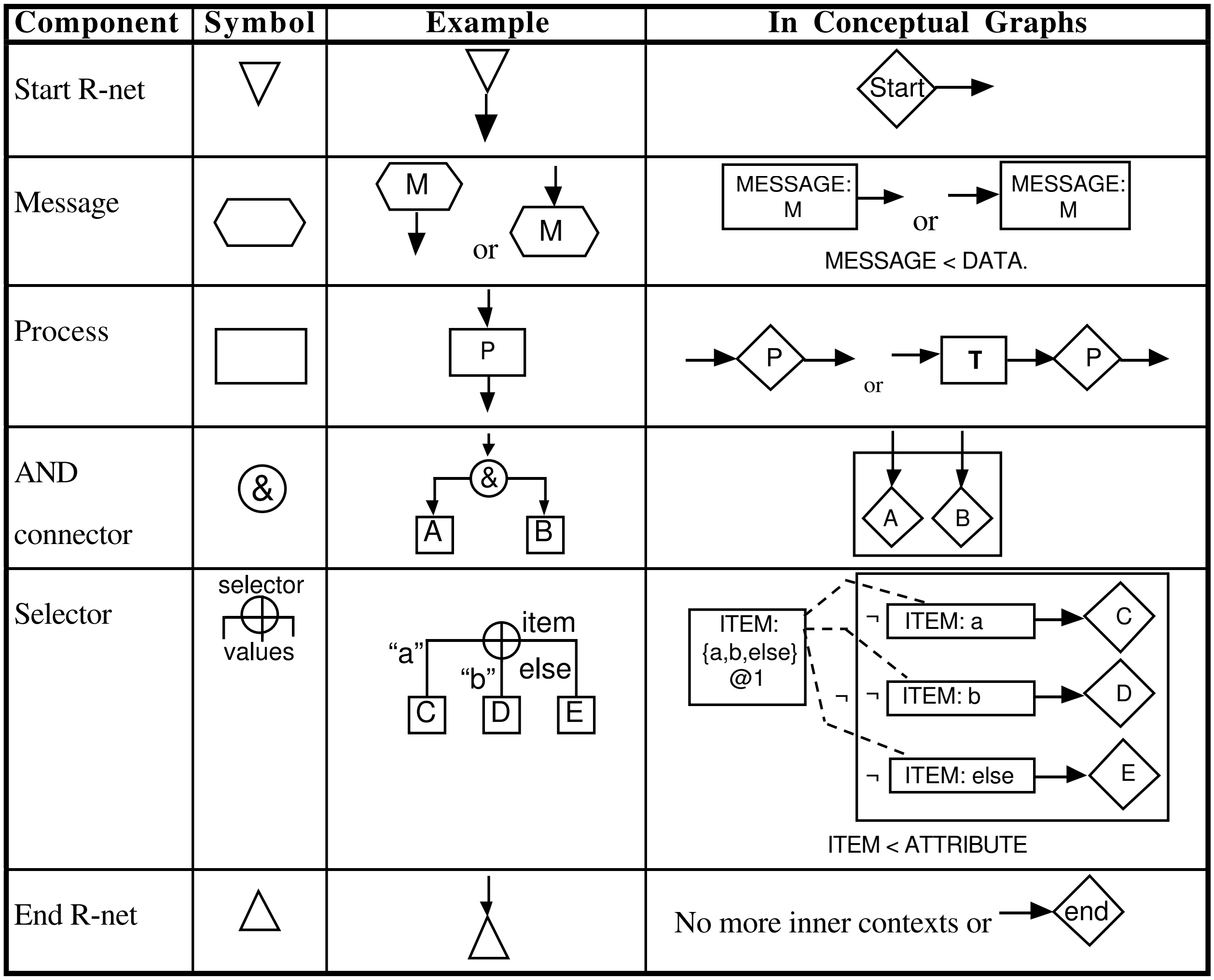
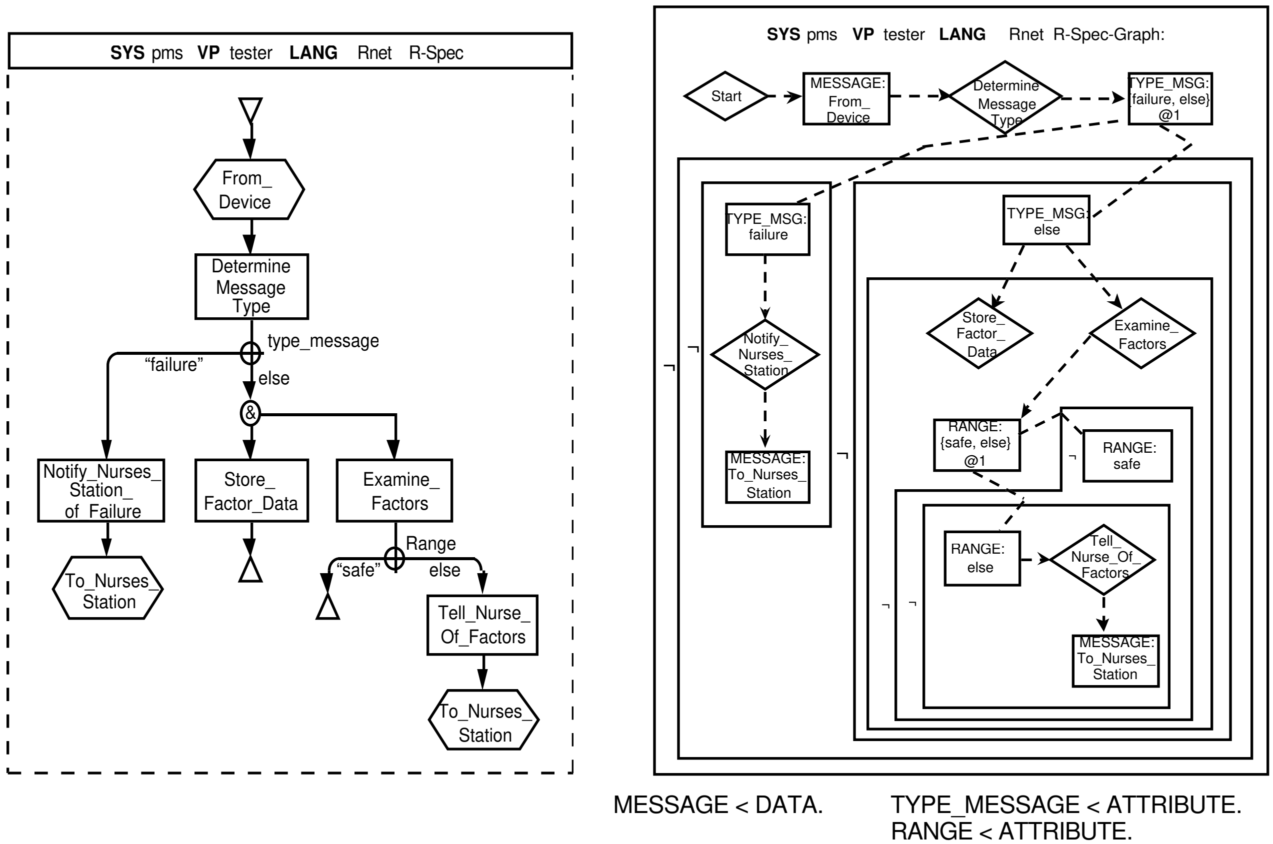


Figure 9. **Translation Of R-net Nodes Into Conceptual Graphs.**



(a) (b)

Figure 10. **Tester's Originating and Translated Requirements Network Diagram.**