Capture, Integration, and Analysis of Digital System Requirements with Conceptual Graphs

Walling R. Cyre, *Member, IEEE*

Abstract—Initial requirements for new digital systems and products that are generally expressed in a variety of notations including diagrams and natural language can be automatically translated to a common knowledge representation for integration, for consistency and completeness analysis, and for further automatic synthesis. In this paper, block diagrams, flowcharts, timing diagrams, and English as used in specifying digital systems requirements are considered as examples of source notations for system requirements. The knowledge representation selected for this work is a form of semantic networks called conceptual graphs. For each source notation, a basis set of semantic primitives in terms of conceptual graphs is given, together with an algorithm for automatically generating conceptual structures from the notation. The automatic generation of conceptual structures from English presumes a restricted sublanguage of English and feedback to the author for verification of the interpretation. Mechanisms for integrating the separate conceptual structures generated from individual requirements expressions using schemata are discussed, and methods are illustrated for consistency and completeness analysis.

Index Terms—Design automation, knowledge acquisition, design representation, knowledge representation, nonmonotonic reasoning, consistency analysis.

1 Introduction

Requirements for new products and systems are generally expressed in a variety of notations that include natural language, diagrams, charts, and tables. While most requirements documents include natural language text, which other notations are employed depends on the type of system or product and to some degree on the enterprise or organization that is preparing the requirements. Although some of the individuals that prepare a requirements document may be trained in a variety of formal notations or languages, all individuals, particularly those having the authority to approve or reject the requirements, are generally not fluent in formal notations, and so, requirements include 'universal' but informal and ambiguous notations such as natural language and diagrams.

A number of projects and systems have been reported for diagrammatic capture of requirements, particularly in the software engineering area. Hardware engineers have long used block diagrams for capture of structural requirements. More recently, several systems have been developed to support capture of behavioral requirements (or models). Process models graphs [3] permit the designer to draw a set of concurrent processes and the signals by which they interact, but the behavior itself must be entered as VHDL code. (The work reported here will avoid this.) The SpecCharts Language [31] is based on diagrams representing concurrent states and their hierarchical decompositions, and are oriented towards control-dominated systems. As with process model diagrams, the activities which occur within a state are specified in a formal hardware description language (VHDL sequential statements). The STATEMATE approach [10] has become the basis for a commercial system [17]. STATEMATE employs a set of three languages State-Charts, Module-Charts, and Activity-Charts to represent the control, structural, and functional aspects of a system, respectively. An analysis of mulit-view, software requirements using conceptual graphs with actors and demons has been reported by Delugach [9]. The notational systems he considers for representing requirements are dataflow diagrams, entity-relationship models, state transition diagrams, and R-Spec graphs.

The goal of the effort reported here is to automatically translate, evaluate, and interpret requirements expressed in natural language and other common notations. The first essential step is translation of the requirements from the various source notations into a common notation so that they may be integrated and analyzed automatically. The notational systems selected for discussion in this paper are natural language (English), block diagrams, flow charts, and timing diagrams. These notations are well known and, together with English, are widely used. For each source notation, the essential concepts and relations that are expressible in the notation are identified, and an algorithm for translating to the common notation is developed. For natural language, the range of concepts and relations has been sharply curtailed to the domain of digital systems. Other notations that have been looked at but are not discussed here because of space limitations include state diagrams, STATEMATE [10], Petri nets, entity-relation diagrams, and dataflow diagrams. A broad selection of notations has been chosen to expose the fundamental concepts and relations used in hardware and software.

Once requirements are entered, translated, and integrated the next step is analysis for completeness and consistency.

• *The author is with the Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0111. E-mail: cyre@vt.edu.*

Manuscript received Jan. 6, 1994; revised Dec. 1, 1994.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96092.

In addition, graphical models are generated for requirements originating in English statements to serve as validation feedback to specification authors of the interpretations of their statements [30]. VHDL engineering models may also be generated from the integrated requirements [15] to provide preliminary design input for further design refinement and subsequent high-level synthesis. The present paper focuses on the capture, integration, and analysis of specifications. (See also [6], [7].)

Because natural language is heavily involved in statements of requirements, a knowledge representation form that readily supports natural language understanding was selected to be the common notation. This notation is a form of semantic networks called conceptual graphs. While a formalism currently used for design representation might have been adopted as the medium for integrating requirements, most formalisms have been designed to represent a limited repertoire of concepts and to support very specific objectives, such as simulation, formal verification, or design synthesis. In mapping information from natural language to a design notation, some information (and all ambiguity) would have to be discarded during translation and before integration. Instead, it was decided to retain the maximum information possible by using a knowledge representation, allowing the integration of requirements derived from more formal notations to resolve ambiguities.

While it is hoped that the results reported in this paper will be broadly applicable, this study focused on specifications and requirements for digital systems. In the sections which follow, conceptual graphs theory is introduced and an overview of the types of concepts compiled for digital systems requirements is given. Conceptual graph theory is based on sets of graphs, called canonical graphs, which are stipulated to be templates of meaningful relationships among concepts in the domain of interest. Each notation used in requirements documents uses a different set of canonical graphs called a canonical basis to express meaning. While the canonical bases overlap strongly in the types of concepts they are constructed from, the relations between concepts vary significantly. For each component (sentence or figure) of a requirements document, an appropriate graph from the canonical basis is selected, instantiated, and joined together to form a representation for the meaning of that component. In this paper, canonical bases and algorithms for generating conceptual graphs are described for a representative set of source notations. Graphs for individual components are integrated by joining together the graphs for components of a requirements document. A method for performing this integration is given. A mechanism that permits tentative integration is considered in order to allow nonmonotonic reasoning through retraction of concept associations which later prove untenable.

Either during the integration process or following integration, the conceptual graphs may be examined for certain types of inconsistencies. In addition, the template nature of the canonical graphs permits the analysis of requirements for omissions by searching for unfilled slots in templates.

2 Conceptual Graphs

Conceptual graphs [28] provide a very useful tool for representing and combining knowledge or meaning. Their power for representing meaning rests in their kinship with semantic networks, while their power for reasoning is derived from their origins in the formal logic (existential graphs) of Peirce [22] and by formal mappings from conceptual graphs to logic formulas.

A **conceptual graph** is a finite, connected, bipartite graph consisting of a set of labeled concept nodes, a set of labeled conceptual relation nodes, and a set of (directed) arcs linking concept and relation nodes.

A conceptual graph consists of at least one concept node, and every relation node is linked to at least one concept node. Each node (concept or relation) has two labels: a type label and a referent label. The set of concept node type labels with the partial ordering, $\leq$, on them forms a lattice called the type hierarchy, where $\perp$ is the universal type and is the absurd type. Similarly, a type hierarchy may be defined on the set of conceptual relation type labels.

In the context of digital system requirements, general concept types include the **device** type which embraces all hardware elements including the subtype **carrier**. The carrier type is used for interconnecting devices such as wires and buses. The **value** type covers the notions of data and messages as well as software. The **action** type is assigned to activities and processes which normally extend over a period of time, whereas the type **event** is typically reserved for discontinuities (beginnings, ends, and interruptions) in actions. The type **condition** encompasses the notions of status or mode as well as statives such as "x *is greater than or equal to* y." Concept types used in measurements include **time** (of durations) and **length** (of data words).

Fig. 1 illustrates a partial hierarchy of the concept types used in this paper for digital systems requirements. The absurd type, $\perp$ which is a subtype of every concept type is not shown in this simplified figure. The type **logic_memory** represents devices that contain both logic for data manipulation and memory for storage of values. As shown, this class of devices contains both processors and counters.

In this paper, a linear notation for conceptual graphs is used. Concept nodes are denoted by two fields separated by colons and enclosed by square brackets, [type : referent]. The first field contains the concept type label and the second field contains a referent marker. The referent markers of concept nodes are used to associate concept nodes with the individuals or abstractions they represent. Common forms of referent labels for concept nodes are shown in Table 1. When the referent marker is absent, the node is assumed to be generic.

Conceptual relation nodes also have two fields, (type: referent), which are enclosed in parentheses. The referent marker of a relation node is used here to contain the preposition indicating the relation if the graph was derived from a natural language expression. (Most workers do not use a referent field for conceptual relation nodes.)

A conceptual graph (in graphical form) representing the statement "the device is reset by an interrupt" is denoted by Graph (1).

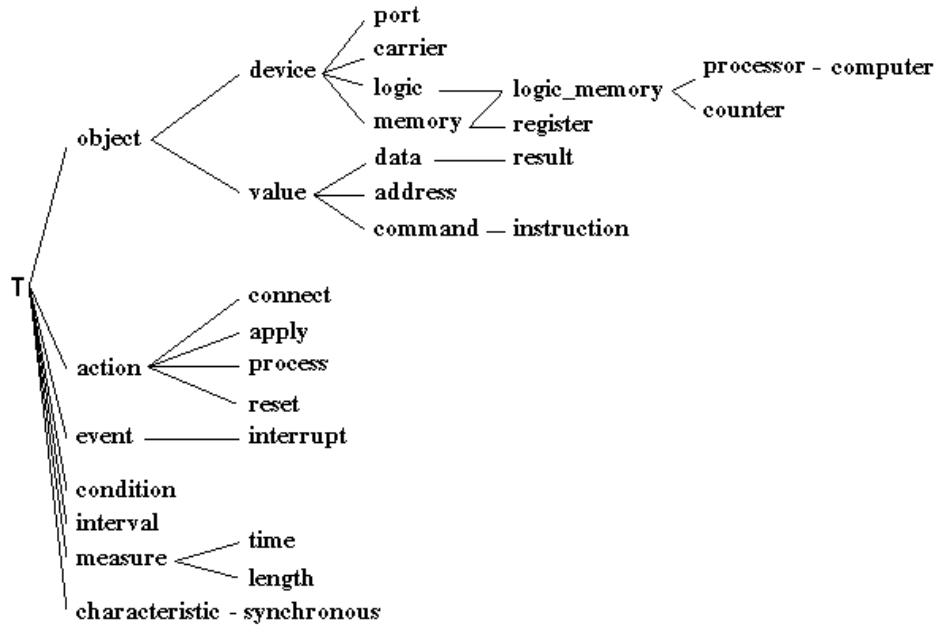


Fig. 1. A partial hierarchy of concept types for digital systems.

TABLE 1
REFERENT MARKERS

Concept	Referent Form	Example
[counter : *]	generic	a counter
[counter : *n]	variable	a counter
[counter : #4n778]	individual	the specific counter with identifier #4n778
[counter : @7]	measure	seven counters
[counter : PC]	named	a counter named 'PC'
[counter : {A, C, *}]	collective set	a set of counters including A and C
[counter : {A | #47 | F}]	disjunctive set	one of the counters A, #47, or F
[counter : ¬]	empty	no counter exists
[condition : G]	conceptual graph	

(1)

This graph indicates that a specific (individual) **device** is the object (**obj**) of the **reset** operation (an action) and some (generic) **interrupt** event is the cause or agent (**agnt**) of the action. The linear notation for this same graph appears as Graph (2).

[reset] –
(agnt : by) -> [interrupt : *]
(obj) ->[device : #]. (2)

In the linear form, only graphs having the form of trees can be represented directly. Graphs having closed walks will have multiple appearances of some concepts. When multiple appearances of a concept are necessary, a common variable denoted by a ***n** is placed in the referent field of each appearance.

In some cases, it will be desirable to indicate possible repetition of relations attached to a concept by the metalanguage symbols **+** and *****. These symbols are not elements in the denotation of conceptual graphs.

[computer] –
+ (part) ->[device],. (3a)

The **+** symbol preceding the part relation in Graph 3a indicates that many **part** relations may be attached to a **computer** concept. An asterisk indicates zero or more repetitions as in (3b).

[string] –
*(part) - > [character],. (3b)

Conceptual relations, such as between which have more than two incident arcs, are denoted with labeled arcs as shown in Graph 4, where 1 and 2 are the arc labels.

[result : x] -> (between) –
1 -> [value : @5]
2 -> [value : @7],. (4)

This graph indicates that a result named x has a value between 5 and 7.

Inverse relations are often encountered, and to simplify the notation, an inverse relation will be denoted by a **–1** following the name of the relation as in **part-1**. The two graphs below are equivalent in representing that some device is the object of a reset action.

[device] -> (obj-1) -> [reset]. (5)

[reset] -> (obj) -> [device]. (6)

Integration of requirements that are represented as conceptual graphs requires a set of operations which may be applied to conceptual graphs to form new conceptual graphs. The four canonical operations defined on conceptual graphs are copy, restrict, join, and simplify. The **copy** operation simply reproduces a graph. The **restrict** operation is used to specialize a concept by replacing its

type by a subtype if the concept is generic, or by replacing a generic referent by an individual referent. For example, [device] might be restricted to [counter: the B timer] in Graph (5). The join operation unites two conceptual graphs on matching concepts. For example, if G is Graph (1), then the graph H = [action]->(manner)->[quickly] can be restricted to

H' = [reset] -> (manner) -> [quickly],
and then, join(G, H') results in Graph (7) below.

[action : is reset] -
(agnt : by) -> [event : interrupt]
(obj) -> [device : the timer]
(manner) -> [quickly],. (7)

In joining nodes with set referents, the set union is taken for collective sets, and the set intersection is taken for disunctive sets. The final canonical operation, **simplify**, deletes any redundant conceptual relations in a graph which resulted from a join operation.

In addition to these elementary operations, some specialized join operations are useful. Two graphs can be joined on multiple pairs of concepts by joining one pair at a time and simplifying the result. A **maximal join** involves the joining of a maximum number of pairs of concepts. A **relational join** joins two relations and joins the corresponding pairs of concepts adjacent to the relation.

A conceptual graph that is meaningful, or represents something that is considered true in the domain of interest, is called a **canonical graph**. The four conceptual graph operations defined above are called canonical formation operations because they can be used to form new canonical graphs from a set of canonical graphs, provided restriction operations obey the type hierarchy and restricted referents conform to their concept or relation type. A set of canonical graphs from which all meaningful graphs in the domain of interest can be formed by the canonical formation rules is called a **canonical basis**. In these terms, each element or primitive in a notation used in a requirements notation is mapped into a graph in a canonical basis for the notation. A requirement, then, is represented by a canonical graph derived from the canonical basis by applications of the canonical formation operations.

As shown later, additional graphs are sometimes needed to integrate requirements derived from different canonical basis. These graphs are called schemata and represent plausible, expanded meanings of concepts. Since schemata describe typical usages of concepts, they are not unique and a concept may have several schemata. A **schema** may be defined in the form below, which gives a schema for the **program_counter** concept type.

schema for program_counter(x) **is**
[counter : ***x**] –
(contain) -> [instruction_address: *a]
(agnt) -> [increment] -> (obj) ->
[instruction_address: *a]
(agnt) -> [load] -> (obj) ->
[instruction_address: *a]
(agnt) -> [clear] -> (obj) ->
[instruction_address: *a]
(part-1) -> [cpu],. (8)

where *x is the formal parameter or variable. This schema states that a (possible) program counter contains an instruction address, which it can increment, load, or set to zero, and that it is a part of a cpu. It should be noted that this schema can also be applied for any other concept it contains, such as cpu and instruction_address. This may not be desirable in all cases, however. For example, the schema includes the concept **increment**, but it is not reasonable to conclude that every increment action is performed by a program counter. A more reasonable schema for increment is given as Graph (9). In a similar manner, schemata for relations may be defined to facilitate integration of conceptual graphs.

schema for increment(x) **is**
[increment : *x] -
(agnt-1) -> [register]
(obj) -> [value: *v]
(contain) -> [value: *v],. (9)

Integration of conceptual graphs involves finding projections of subgraphs within other graphs. A **projection**, π, is a mapping of one conceptual graph, u, into another, v, such that πv is a subgraph of v and for every concept c in u, πc is in v and type(πc) $\leq$ type(c), if the referent of c is individual then πc is identical, and for every relation r in u, πr is in v and type(πr) = type(r). Although finding subgraphs is generally an NP-complete problem, in the present case the conceptual graphs to be projected can be converted into trees by replicating nodes. Mugnier and Chein [21] have shown that finding projections of a conceptual tree into a conceptual graph is polynomial on the order O(cr), where c is the number of concept nodes of the tree and r is the number of relation nodes of the graph. In the present application, the conceptual trees (subgraphs of schemata) are generally quite small.

3 CAPTURE

The first step in the automatic evaluation and interpretation of requirements is their capture, which consists of entry and translation to conceptual graphs. Each source notation for requirements has an editor which supports the entry of expressions of requirements. The difficult step is the translation of source expressions to conceptual graphs, particularly expressions in natural language. In the following sections, a selection of notations used in requirements and specification documents will be considered. The notations of block diagrams, flowcharts, timing diagrams and natural language have been chosen here to provide a wide span of semantics in the hardware and software domains. For each of these notations, a canonical basis is proposed and an algorithm for generating conceptual graphs from expressions in these notations is described. After introducing the notations and their translation, the integration of requirements using the formation operations and schemata will be presented.

3.1 Block Diagrams

Block diagrams are frequently used to illustrate structural requirements, such as how components are intercon-

nected or which are subcomponents (parts) of others. Such diagrams can consist of four basic elements: blocks, lines, attachments, and labels. The blocks depict hardware components and therefore correspond to **device** type concepts. Lines or arrows typically indicate interconnections, which correspond to **carrier** concepts. (In the nomenclature of schematic diagrams, carriers are generally called nets.) The points of incidence between lines and blocks, or attachments, constitute the third basic element of block diagrams. These incidence points are called **ports**, and are mapped into a third basic type of concept. Physically, ports may be associated with pins, and each port may represent one pin or a vector of pins. Directions of information transfer are typically associated with ports (input ports and output ports), but to be more precise here, the direction of transfer will be represented by a conceptual relation between the port and the device it is attached to. These attachment relations are listed in Table 2. For example, an input pin of a device has an input relation (<<) between the device and the port, as in [device] -> (<<) -> [port]. Directionality may be fully (<<, >>, <>) or partially (<-, ->) specified. The relation (–) indicates an attachment with no specification of directionality, and (<>) specifies a bidirectional relations such as between all ports and carriers.

TABLE 2
BLOCK DIAGRAM ATTACHMENT RELATIONS

symbol	relation
–	attachment, direction unspecified
<-	input and possibly output
->	output and possibly input
<<	input only
>>	output only
<>	bidirectional

These six conceptual relations between devices and ports obey the type hierarchy shown in Fig. 2.

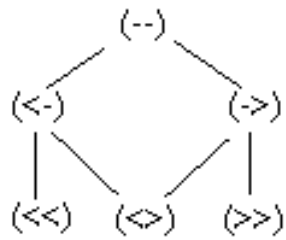


Fig. 2. Hierarchy of directionality relations.

Since the connection of a carrier to a port of a device is nondirectional, the bidirectional relation (<>) is used to denote attachments of carriers to ports. The canonical basis for block diagrams consists of two conceptual graphs. Graph (10) is the canonical graph for a device and consists of one device concept and one or more ports. The nondirectionality of the attachment relation is expected to be restricted when the graph is instantiated in a requirement.

```
[device] -
        +(–) -> [port],.        (10)
```

The canonical graph for a carrier is given as Graph (11). A carrier typically connects two or more devices, so an isolated carrier should not occur. By limiting the canonical graphs in this way, some requirements errors can be eliminated during the translation process by not accepting some constructs in the source notation.

```
[carrier] -
        +(<>) -> [port] <- (–) <- [device]        (11)
```

In addition, the form of Graph (11) is useful in translating netlists, a common representation of block diagrams, to canonical graphs. Netlists can be derived automatically from block diagrams by schematic capture tools used in design automation. The Electronic Design Interchange Format (EDIF) is a standardized netlist form [29]. In the LISP-like format of EDIF, each construct is enclosed in parentheses and consists of a keyword followed by a list of parameter values or constructs, as (cellRef A) declares that A is the identifier of a type of device (a cell). An algorithm for deriving a conceptual graph from a (simplified) EDIF representation is given below. A number of mandatory format levels of EDIF have been left out here to simplify the present discussion.

Algorithm: Conceptual Graphs from EDIF Netlists

1) a) For (cell A (interface (port P (direction D))))
      generate [A] -> (D) -> [port : P]
   b) For (instance X (cellRef A))
      generate [A : X]
   c) For (net C (joined (portRef Pi (instance Xj))))
      generate [carrier : C] -> (<>) -> [port : Pi] <- (- -)
      <- [device : Xj]
2) Restrict each device concept in a carrier conceptual graph generated by 1c) to the type specified in the device's instance declaration.
3) Restrict each relation between a port and device in a carrier conceptual graph by 1c) to the directional type specified in the device's cell declaration.
4) Join the individual graphs generated from the netlist and simplify the result.

The first step of the algorithm generates all the concepts (devices, carriers, and ports) and their interrelationships. The second step restricts the general type **device** in nets to the subtype identified in the instance declaration, and the third step restricts the relations between devices and ports to the directionality specified in the cell declaration. With this preparation, the device graphs and carrier graphs will join properly on identical device-relation-port subgraphs. Redundant copies of relations are then eliminated by the simplify operation.

The above accounts for the interconnections in a block diagram but does not account for the hierarchical composition of devices by their components. Graph (12) is the basis graph for accommodating this information by showing that a device has zero or more components related to it by a conceptual relation of type **part**.

```
[device] -
        *(part) ->[device],.        (12)
```

Containment information is also included in netlist notations, such as EDIF, through recursive data structures.

Whenever a device is declared in the beginning of a netlist, its substructure as well as its ports may be declared. This substructure is readily captured by instantiating Graph (12) in an expanded form of the preceding algorithm.

3.2 Flowcharts

Although flowcharts of programs and protocols are graphically similar to block diagrams and can employ the EDIF notation for their machine representation, they are semantically very different and, therefore, require a distinct canonical basis. The primary semantic elements of flowcharts are processes (rectangles), terminals (rounded rectangles), decisions (diamonds), and flowlines. In the present discussion, it is assumed that flowcharts describe only uniprocessing situations and are, therefore, free of parallel lines of computation. The process blocks of flow charts are readily modeled by action concepts. Flowlines may have any number of sources (antecedent processes) but only one destination process (consequent process). It is further assumed that an antecedent process finishes when the consequent process starts, so the conceptual relation holding between two such processes is finishes_when_starts (fws) as in [action] -> (fws) -> [action]. To facilitate integration with graphs from other notations, however, it is preferable to use the inverse relation **starts_when_finishes** (**swf**) in the canonical basis for flowcharts since this is more consistent with English expressions of temporal relations. The canonical graph for a simple flowline appears as Graph (13a), where action_2 is the antecedent process and action_1 is the consequent process.

[action_1] -> (swf) ->[action_2]. (13a)

Flowlines having multiple sources, such as occur with loops and reconvergence of flow paths after decisions, are or-joins of flowlines, and require the introduction of a disjunctive action concept as in Graph (13b). Action_2 in this graph represents either action_3 or action_4. Note that the or relation here is ternary, and similar canonical graphs are needed to cover flowlines having various numbers of sources.

```
[action_1] -> (swf) ->
   [action_2] -> (or) -
                    1 -> [action_3]
                    2 -> [action_4],.
```
(13b)

Semantic modeling of decisions in flowcharts is more complex. Although the graph [action] –> (swf) –> [decision] might be considered in representing decisions in flowcharts, this does not readily accommodate expression of the decision outcome associated with the flowline. The decision outcome could be placed in a referent field of the **swf** relation, but this does not integrate easily with graphs from other notational sources. Instead, a decision here is viewed as a set of flowlines which pass through the decision symbol; each flowline is associated with an outcome of the decision. Therefore, a flowline passing through a decision will be related to one antecedent action and one condition, as well as the consequent process. The condition concept represents the outcome of the test which enables the consequent process.

```
[action_1] -> (swfif) -
                1 -> [action_2]
                2 -> [condition],.
```
(14)

The core of this canonical graph is the **starts_when_finished_if** (**swfif**) relation which is interpreted: action_1 starts when action_2 finishes if the condition is true. In many cases, decisions are labeled by a variable inside the diamond, and a value for the variable on each flowline passing out of the decision. Such conditions are captured by canonical graph (15). Relations such as greater than of equal, less than, and the like may also be used to express conditions. Condition concepts can accept entire conceptual graphs as their referents, that is, a condition concept refers to a conceptual graph like (15) as illustrated by Graph (16).

[variable] -> (=) -> [value]. (15)

[condition : [variable : F] -> (=) -> [value : True]]. (16)

When a flowline passes through a sequence of decisions, the enabling condition is the conjunction (AND) of the individual conditions. This may be represented using **part** relations for an AND conjunction as in Graph (17).

```
[condition] -
        (part) -> [condition]
        (part) -> [condition],.
```
(17)

Although terminals often only denote continuations of flowlines in partitions on flowcharts, they may also serve to identify the first and last subprocesses that are executed as part of an encompassing process. Therefore, as entry and exit points it is useful to model terminals by attaching unary conceptual relations on the subprocesses that are entered into or exited from. To support this the following two graphs are added to the canonical basis. Note that the entry and exit terminals are expected to have identifiers, #t, to fill the referent positions in the **entry** and **exit** conceptual relations.

[1: action] -> (entry: #t). (18)

[1: action] -> (exit: #t). (19)

The final graph of the conceptual basis for flowcharts supports the joining of disconnected flowcharts, by establishing **swf** relations between processes whose connecting flowlines are interrupted by terminals.

```
[action] -
        (entry : *t)
        (swf) -> [action] -> (exit : *t),.
```
(20)

The algorithm given here for generating conceptual graphs to represent flowcharts, first requires expanding each decision into an exhaustive collection of condition-labeled flowlines between processes, and then expanding all multiple flowlines. Then, a basic canonical graph can be generated from each flowline, and these conceptual graphs can be integrated by join operations. Finally, flowlines interrupted by terminals can be completed by searching for possible joins using Graph (20).

Algorithm: Conceptual Graphs from Flowcharts

1) Treating Decisions:
   a) Replace every IF-THEN decision by two condition-labeled flowlines.

Replace every n-way CASE decision by n condition-labeled flowlines.

b) Generate a condition graph, Graph (15), from each flowline condition label, also using Graph (17) where flowlines are labeled by multiple conditions.

2) Treating Flowlines:

a) For every unconditional flowline between two apply Graph (13a).
For every multiple-source unconditional flowline, apply Graph (13b) or a graph with a suitable (or) relation.
For every conditional flowline between two processes, apply Graph (14) using the condition graph as the referent of the condition concept in Graph (14).

b) For every flowline staring from an entry terminal, label the destination action concept by an entry concept with that label as its referent.
For every flowline terminating in an exit terminal, label the source action concept by an exit concept with that label as its referent.

c) Join Graph (20) with every pair of processes having respective exit and entry terminals with the same referents.

Since flowcharts are readily modeled as graphs, any graph search procedure for cyclic graphs can be used to examine all elements of a flowchart. As each decision or flowline is examined, the algorithm above is applied to generate the conceptual graph. This algorithm deals with a common style of flowcharting. Other flowchart styles may require modification of the canonical basis as well as the algorithm.

3.3 Timing Diagrams

Timing diagrams are often used in requirements and specifications to illustrate system behavior in the form of simulation results (although the simulation is generally performed mentally). A timing diagram consists of a set of parallel timelines, each associated with a particular carrier (line or bus). Each timeline is partitioned into intervals during which a value (**obj**) is applied to the carrier (destination) by some device (instrument) attached to the carrier. (The value applied to the carrier may be indeterminate or undefined.) Each interval on a timeline can be represented by a concept node. To permit the interval to be identified with a carrier, a device, a value, and a duration (**dur**), the type definition below may be used.

```
type interval(x) is
        [apply : *x] -
                (inst) -> [device]
                (obj) -> [value]
                (dest) -> [carrier]
                (dur) -> [time],.                    (21)
```

When a timing diagram is viewed as a parallel set of sequences of intervals, temporal relations from systems of interval temporal logic [2], [11], [20] can be employed as conceptual relations between the intervals. The interval relations used here are the endpoint relations defined by Matuszek, which are used to define the temporal relations between the endpoints (beginnings and ends) of pairs of intervals. The set of seven endpoint relations (with their inverses) listed Table 3 are adequate for describing timing diagrams.

TABLE 3
TEMPORAL INTERVAL RELATIONS

Endpoint Relation	Conceptual Graph	Inverse Relation
starts before starts	[interval] -> (sbs) -> [interval]	sas
starts when starts	[interval] -> (sws) -> [interval]	sws
starts before finishes	[interval] -> (sbf) -> [interval]	fas
starts when finishes	[interval] -> (swf) -> [interval]	fws
finishes before starts	[interval] -> (fbs) -> [interval]	saf
finishes before finishes	[interval] -> (fbf) -> [interval]	faf
finishes when finishes	[interval] -> (fwf) -> [interval]	fwf

As an example, if interval x starts before interval y finishes, [x] –> (sbf) –> [y], then the inverse is true, i.e. interval y finishes after interval x starts, [y] –> (fas) –> [x]. Two of the above relations (**sws** and **fwf**) are self-inverses. Note in particular that the **starts_when_finished** (**swf**) conceptual relation is the same one used in generating conceptual graphs from the flowlines of flowcharts. This will be of use in integrating requirements from flowcharts and timing diagrams.

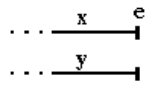
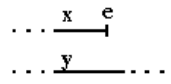
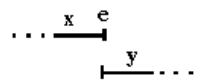
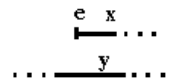
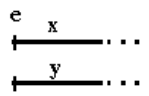
Generating conceptual graphs from timing diagrams using a larger set of temporal relations has been described [8] in some detail in another paper, so a simplified but adequate generation algorithm is presented here for this set of relations. This algorithm is less efficient and generates more relation nodes in the resulting graph. The present generation algorithm depends on analyzing the configuration of intervals at points in the timing diagram where at least one interval ends (and necessarily another begins). Consider one such event, e, in a timing diagram. Three sets of intervals can be defined with respect to this event. Let F denote the set of intervals that finish at the event. Let S denote the set of intervals that start at the event, and finally, let set C contain the intervals that continue through the event without interruption. It may be noted that $|F| = |S|$, $|F|+|C| = |S| +|C| = n$, where n is the number of timelines. ($|Q|$ denotes the cardinality of set Q.) Table 4 shows the canonical graphs that describe configurations of pairs of intervals from these three sets. For example, when one interval is taken from S (starts at e) and the second interval continues through the event (is in set C), then Graph (22d) is generated.

The graphs in Table 4 together with Graph (21) form the canonical basis for timing diagrams. An algorithm for generating conceptual graphs from timing diagrams is given below.

Algorithm: Conceptual Graphs from Timing Diagrams

1) Let C initially contain every interval that begins a timeline. Let F and S be empty.
Let G be the empty graph.
2) For every interval $i \in C$ join [interval : i] to G.
3) For every event, e, along the timing diagram
Let F contain the intervals that finish at e,
Let S contain the intervals that start at e,

TABLE 4
CANONICAL GRAPHS FOR TIMING DIAGRAMS

Intervals		Canonical Graphs	
$x \in F$ $y \in F$		[Interval : x] - (fwf)->[Interval : *y] (fas)->[Interval : *y] (sbf)->[Interval : *y]	(22a)
$x \in F$ $y \in C$		[Interval : x] - (fbf)->[Interval : *y] (fas)->[Interval : *y] (sbf)->[Interval : *y]	(22b)
$x \in F$ $y \in S$		[Interval : x] - (fws)->[Interval : *y] (sbs)->[Interval : *y] (fbf)->[Interval : *y] (sbf)->[Interval : *y]	(22c)
$x \in S$ $y \in C$		[Interval : x] - (fas)->[Interval : *y] (sas)->[Interval : *y] (sbf)->[Interval : *y]	(22d)
$x \in S$ $y \in S$		[Interval : x] - (sws)->[Interval : *y] (fas)->[Interval : *y] (sbf)->[Interval : *y]	(22e)

Let C = C – F.
For $x \in S$, join [interval : x] to G,
For $x \in F$, $y \in F$, $x \neq y$, generate Graph (22a) and join to G,
For $x \in F$, $y \in C$, generate Graph (22b) and join to G,
For $x \in F$, $y \in S$, generate Graph (22c) and join to G,
For $x \in S$, $y \in C$, generate Graph (22d) and join to G,
For $x \in S$, $y \in S$, $x \neq y$, generate Graph (22e) and join to G,

4) Simplify G.
5) Repeat Steps 2 and 3 until all events have been scanned.
6) Expand each interval concept in G using Graph (21).

This algorithm is not highly efficient since multiple copies of relations may be introduced and later eliminated in Step 4, but the algorithm will capture the necessary relationships. To keep the conceptual graphs generated by this algorithm fairly sparse, the algorithm does not calculate the endpoint relationships between all pairs of intervals in a timing diagram. Relationships between other pairs of intervals can be computed using transitivity tables reported elsewhere [2], [8], [11], [20]. For example, if [x] -> (fbf) -> [y] and [y] -> (fws) -> [z], then it is true that [x] -> (fbs) -> [z].

Timing diagrams often have annotations indicating delays or durations between events. As shown in [8], such annotations can be captured in conceptual graphs using additional conceptual relations. For example, let (safby) represent the ternary relation starts_after_finishes_by as in Graph (23), which indicates that interval x starts after interval y finishes by t seconds.

```
[interval : x] -> (sasby)
                  1 -> [interval : y]
                  2 -> [time : t seconds],.        (23)
```

3.4 Natural Language

The bulk of many requirements documents, contracts and proposals consists of natural language expressions. While the automatic analysis of standard or free English is a major unsolved problem, progress has been made in the analysis of sublanguages of English [18] which have a limited vocabulary and grammar and cover a restricted semantic domain. Semantic domains which have received substantial attention include those characterized by medical reports, legal documents, message systems, and news reports. Relatively little effort has been invested in the automatic analysis of specifications on digital systems, and much of that has been focused on executable software specifications and natural language programming. An important example of the automatic analysis of natural language specifications on digital systems is the work by Granacki and Parker [13], which employed semantic patterns (a semantic grammar [4]) in a phrasal analyzer called PHRAN to generate conceptual dependency structures [25] from English sentences. The conceptual dependencies were then transformed into

formal specifications expressed in a design data structure (DDS) [19] was then generated from by a specification analyzer (SPAN). The conceptual dependency structures served the role of conceptual graphs in the present approach. Conceptual dependency theory is a strong slot and filler structure [24] with a small set of primitives, whereas conceptual graphs are a weak slot and filler structure with a large canonical basis. As a result of having a small canonical basis, representations in conceptual dependencies can become quite complex because everything must be decomposed in to a small set of simple primitives, whereas, the large canonical basis used in conceptual graph theory affords more economical representations. This is an important consideration in subgraph searches to perform integration of representations.

The approach described in this paper employs compositional semantics. First, syntax analysis of requirements statements is performed by a parser. The resultant parse trees are then passed through a semantic analyzer. Syntax analysis is performed by a bottom-up parallel chart parser [1], [32] with a phrase-structured grammar. This grammar has little more than a hundred rules, but is rich enough to accept complex sentences such as the ones listed below.

In limiting the grammar rules, selections have been made to eliminate some of the more troublesome constructions used in standard English. For example, the use of conjunctions has been restricted, and since interrogative (question) and imperative (command) sentences do not occur in specifications, these are not covered by the grammar. The sublanguage used in specifications does have some constructions that are not generally used in standard English, such as using the infinitive forms of verbs as nouns, as in “a subsequent write to memory by the processor.” Normally, this would be expressed using a gerund as in “a subsequent writing to memory by the processor.” Because even this restricted sublanguage of English is ambiguous, the parser will produce one or more parse trees for each sentence accepted by the grammar. In each parse tree, the leaves of the tree are the terminals (words) of the language, and the remaining nodes represent grammatical constructs such as noun phrases, verb phrases and clauses. The root of a tree represents the sentence construct.

TABLE 5
EXAMPLE REQUIREMENTS SENTENCES

Stacking the cpu registers, setting the low bit and vector fetching requires a total of 11 tcyc periods for completion.

The remaining cpu registers are not pushed onto the stack.

The machine can perform a read or write asynchronously if the W/R signal is high.

The -int pin can also be polled with branch instructions.

The 8048 has 27 lines which can be used for input functions or output functions.

The 8-bit data is loaded into the -acc register when -strb rises.

Buffen is high when -ds1 is high and -nds2 is low.

The data in -acc is applied to -do when buffen is high.

The i80486 contains a cache, processor and co-processor.

Resetting before the system stops deletes the startup program.

TABLE 6
CONCEPTUAL RELATIONS FOR ACTION AND EVENT GRAPHS

acc	accompaniment
agnt	agent
attr	attribute
dest	destination
inst	instrument
man	manner
obj	object
ord	ordinal
purp	purpose
quant	quantity
src	source

The function of the semantic analyzer is to select the most meaningful parse tree if more than one occurs, and to generate a conceptual graph that reflects the meaning of the sentence. Each word carries meaning or performs a function which is represented by one or more conceptual graphs in the canonical basis. Because some words are synonyms of others (clear, reset) and several forms may have the same basic meaning (write, writes, writing, written) the canonical basis for English will be large, but not unmanageable. Some words have more than one meaning, but unlike general English in which most words have several meanings, most words in requirements have only one or two meanings resulting in a relatively small canonical basis. Examples of canonical graphs for a few English words will be introduced as needed in the following discussions. A selection of conceptual relations used in the conceptual basis for English is tabulated below. These conceptual relations for actions and events are based on the case frames by Fillmore [12].

The algorithm for semantic analysis described here is similar to that reported by Sowa and Way [27]. For a given parse tree of a sentence, each leaf of the parse tree corresponds with a sentence word and is replaced by the word’s canonical graph from the basis. Unrecognized character strings are interpreted as names (identifiers or acronyms). In each of these canonical graphs one concept is called the head concept of the graph, generally the concept most closely associated with the sentence word. For each nonleaf node, a conceptual graph is constructed from the graphs associated with its daughter nodes in the tree. One of the daughter nodes is considered the head and the remainder are considered to be modifiers. First, the head node’s conceptual graph is raised to the current node, and its head concept becomes the head concept of the current node. Next, the graph of each modifier is attached to the current node’s graph. In a data base, a semantic analysis rule exists for each grammar rule. Since each nonterminal node of a parse tree is generated by a grammar rule, the associated semantic analysis rule is chosen to identify the head daughter node and to govern the joining of modifier graphs to the head graph. For example consider semantic analysis of the noun phrase “the synchronous counter” represented by the subtree (np (det the) (adj synchronous) (noun counter)) where the canonical basis includes the following three graphs:

the	[T : #].	(24)
synchronous	[device] -> (attr) -> [synchronous]	(25)
counter	[counter].	(26)

The noun is the head node of this noun phrase so Graph (26) is raised to the noun phrase node. Next, the canonical graph for "the" is joined to the graph for counter producing (27). In order to perform this join, the universal type T must be restricted to the subtype **counter**. Note that this referent indicates that a specific individual is referred to but the individual's identifier is not given.

```
[counter: #].                                              (27)
```

Then, the canonical graph for "synchronous" (25) is retrieved and joined with the current graph (26) after restricting **device** to **counter**, resulting in Graph (28).

```
[counter: #] -> (attr) -> [synchronous].                   (28)
```

This example was quite straightforward, but the semantic analysis rules for some grammatical constructs can be quite complex, and may be sensitive to the concept types of the head and modifier concepts. As a more complex example, consider the grammatical predicate "writes data to memory" having the subtree (pred (active_verb writes) (noun data) (pp (prep to) (np (det a) (noun register)))). The relevant canonical graphs are listed below.

```
write    [write] –
                 (agnt) -> [action]
                 (obj) -> [value]
                 (src: {from}) -> [device]
                 (dest: {to | into}) -> [memory],.         (29)
data     [data].                                           (30)
to       [action] -> (T: to) -> [object].                  (31)
a        [T: *].                                           (32)
register [register].                                       (33)
```

The problem here is that **register** is a subtype of both device and memory, so joining the graph for "register" to the graph for write is nondeterministic. To overcome this, a relational join is performed. In processing the predicate, the semantic rule first raises the graph for "write." Next, the graph for data is joined to the object concept predicated on its syntactic role of being the direct object of the predicate, and under the constraint that **data** is a subtype of **value**. Since the next constituent is a nonterminal (prepositional phrase), it is processed to produce Graph (34) from the graph for the preposition and the graph for the noun phrase.

```
[action] -> (T: to) ->[register: *].                       (34)
```

Notice that the preposition appears as an individual referent in its canonical graph and not as a concept. In order to perform a relational join between Graphs (29) and (34), the universal relation must be restricted to the **dest** relation, and the individual referent "to" must be coerced into a singleton disjunctive set referent as in (35).

```
[action] -> (dest: {to}) -> [register : *].                (35)
```

Now, (29) and (35) can be joined to produce the final conceptual graph (36) below.

```
[write] -
        (agnt) ->[action]
        (obj) ->[data]
        (src: {from}) ->[device]
        (dest : {to}) ->[register : *],.                   (36)
```

Even though only a sublanguage of English is processed by the grammar and semantic analyzer, it is sufficiently complex to preclude a detailed description here. Instead a few examples of natural language statements and the conceptual graphs they generate are given here to support the discussion of integration of requirements which follows.

Block diagrams generally convey interconnection of devices and hierarchical arrangements of devices. A natural language expression for interconnection is, "Device X is connected to device Y by carrier D." Graph (37) below is generated from this expression.

```
[connect] -
        (inst) -> [carrier: D]
        (obj) -> [device: X]
        (dest: {to}) -> [device: Y],.                      (37)
```

Hierarchical structure may be expressed in English by expressions of the form, "Device X consists of device B, device C and device D," which generates Graph (38).

```
[contain: consists of] -
        (agnt) ->[device : X]
        (obj) ->[device : AND] -
                (part) ->[device: B]
                (part) ->[device: C]
                (part) ->[device: D],.                     (38)
```

Expressions which convey relationships comparable to those expressed in flowcharts include "If X is Y, then the register is reset." Conceptual graph (39) is generated from this form of statement.

```
[reset] -
        (obj) ->[register: #]
        (if) ->[condition: [be: is] -
                (agnt) ->[variable: X]
                (obj) ->[value: Y],.],.                    (39)
```

Another representative expression is the statement of causality: "action causes action," such as "Executing a CALL instruction causes the address to be saved."

```
[cause: causes] -
        (agnt) -> [action: execution] -> (obj) ->
                [instruction: *] - > (name) -> [CALL]
        (obj) ->  [action: to be saved] ->
                (obj) ->[address: #],.                     (40)
```

Quite often this type of relationship is expressed more succinctly by letting the causative action be the agent of the consequent action as in, "Executing a CALL instruction saves the address," which has the graph below.

```
[action : saves] -
        (agnt) ->[action : execution] ->(obj) ->[instruction :
                *] -> (name) ->['CALL']
        (obj) ->[address : #],.                            (41)
```

Temporal relationships are expressed in natural language through subordinating conjunctions (s_con) such as before, after, while, during, until and concurrently with. These are used in the form "action conj action," as in "The processor waits until a message is received," and generates a conceptual graph of the form

[action] -> (t_rel: {s_con}) -> [action] (42)

where t_rel is a temporal relation and s_con is a subordinating conjunction. Subordinating conjunctions are not exact terms, but for this discussion, they will be interpreted as tabulated in Table 7.

TABLE 7
TEMPORAL RELATIONS FOR SOME SUBORDINATING CONJUNCTIONS

s_con	t_rel	Comments
after	saf	starts after finished
before	fbs	finishes before starts
concurrently with	fas, sbf	two endpoint relations
during	sas, fas, fbf, sbf	four endpoint relations
then	sbs	starts before starts
until	fws	finish when starts
when	sas, sbf	two endpoint relations
while	sas, fas, fbf, sbf	same as during

These cases provide only a very small sample of the rich variety of English expressions which may convey information corresponding with the information expressed in more formal notations. It will be noticed that the conceptual graphs of the English expressions do not correspond directly with graphs derived from comparable expressions in the more formal notations. To overcome this, methods for joining appropriate conceptual graphs from different notations are given in the next section.

4 INTEGRATION

As a requirements document is prepared, each component of the document (a figure or sentence) will generate a separate conceptual graph representing its meaning. To exploit the benefits of the common notation of conceptual graphs, it is necessary to integrate the requirements by joining these separate conceptual graphs. As described earlier, two conceptual graphs (derived from distinct requirements expressions in a specification) may be joined on identical concepts or identical subgraphs. Two concepts which are not identical can often be made identical by restriction operations. Similarly, if a (small) conceptual graph can be projected into two other conceptual graphs, then the two graphs may possibly be joined on the projected images. Such joins may be made maximal by extending the common subgraphs being joined in both of the original graphs as far as possible. While many subgraphs may be found in two conceptual graphs to support joins, only a few of these may be desirable, since joins should only be made on concepts (or subgraphs) which have the same referent, or mean the same thing.

Detection of common references to a particular individual in informal notations is a major unsolved problem, particularly in natural language where it is referred to as anaphora resolution. In formal systems, such as hardware description languages, this problem is avoided by declarations which force the designer to name every object and value by a unique identifier. Restricting the informal notations and English sublanguage used here to this practice would destroy the value of these notations in allowing the user to focus on ideas and relationships rather than on details such as names and identifiers. Requiring an identifier for every concept in the restricted sublanguage used for requirements would also make it less readable, as in the example "the remaining cpu (#45) registers (#589) are not pushed (#2388) onto the stack (#8334)." This technique is used to some extent in U.S. Patents, though not for automatic processing.

In considering the integration of concepts referring to common individuals, it is clearly appropriate to join two concepts which have identical, individual markers (identifiers) in their referent fields. Beyond such exact matches, however, joining pairs of similar concepts is not always well advised. For example, in the context of uniprocessors, joining the **counter** concepts in [counter: #] -> (name) -> ['program counter'] and [counter: *] -> (name) -> ['program counter'] is reasonable since a uniprocessor has only one program counter, whereas joining [register: *] with [register: #] is generally not advisable unless, perhaps, they occur in precisely that order in the same or consecutive sentences of narrative with no intervening references to registers. Several strategies are available to deal with the common reference problem. Where such joins are possible, they might be decided manually by a technical expert. This is the least desirable option. Second, a query to the author of the requirements might be generated for confirmation of a join hypothesis. Third, the join might be decided on some measures of confidence such as their being in consecutive sentences or the same paragraph, or their being of a singular type, such as "accumulator." In practice, all of these mechanism may be employed in the appropriate circumstances. Except when the joins are directed by the requirements author or involve unique identifiers, they will involve some uncertainty or abductive reasoning [14] in that the concepts joined are hypothesized to have the same referent based on their type and relations with other concepts. Therefore, it may be desirable to retract them later if contradictions do occur. But once concepts are joined, their individual forms are lost, so the simple join only supports monotonic reasoning. To permit nonmonotonic reasoning where "joins" can be retracted later, consider an approach using tentative or retractable joins. Let a **tentative** join be denoted by a **same** relation between the two concepts involved, so that the graph

[type_x] -> (same) -> [type_x]

indicates that both concepts are assumed refer to the same individual. If a new concept is joined with one of two tentatively joined concepts, then a **same** relation must be established with the other concept as well. In general, a set of concepts joined by the **same** relation must be a complete subgraph, and the **same** relation is its own inverse. The approach of tentative joins can be extended for approximate reasoning by defining an ordered set of relations having different degrees of sameness, or by using the referent field of the **same** relation to indicate the level of confidence of the relation. Such quantified **same** relations are useful where the position of the requirement in the document or the distance between their sources in the document are considered in deciding on a join. Another major advantage of using **same** relations rather than simply joining concepts is that the individual graphs can be identified and traced back to their source notational units to facilitate back annotation,

a very attractive feature in supporting integrity between requirements documents and products derived from them. The disadvantage, of course, is that more computational effort will be required in comparing and operating on complete subgraphs of **same** related concepts rather than on single concepts.

In a related project [26] to develop an automatic coreference detector for object (device and value) concept types, good results have been achieved. The Coreference Detector developed in that work is a rule-based system that maintains a table of references to objects encountered in a text. These references are called definitions. When a subsequent object reference is encountered, it is compared with entries of comparable type in the table and classified as a coreference or definition based on the system of rules. In 80%-95% of the cases, coreferences are properly classified. When errors occur, the program conservatively misclassifies references as definitions rather than as coreferences.

As an alternative to these tentative joins, the lines of identity or coreference links defined by Sowa [28] can be used to associate concepts. Coreference links are arcs between multiple appearances of a given concept, and are necessary in nested conceptual graphs. These links are not labeled and do not involve relations. In addition, coreference links denote equality of concept appearances and, though they, like any conceptual graph structure, might be retracted, there is no label that can be used to indicate the degree or confidence level of sameness or equality.

Next, the effects of differing canonical bases on the integration of requirements must be considered. Even with tentative joins, the desired integration of requirements from different source notations (text and diagrams) is likely to remain undesirably low because the various source notations employ different canonical bases. The cost of failing to join two concepts that refer to the same individual is that two distinct individuals of that concept are implied. Thus, the apparent cost of the system will be higher (multiple device concepts) or performance will be lower (multiple action concepts). Concepts and relations are sites for integration of conceptual graphs. So, if concepts and relations implied by subgraphs are added to a graph, then the potential for integration with other graphs is increased (at the cost of searching on the increased number of nodes). The augmentation of graphs by implied concepts and relations might be expressed as rewrite rules of a graph grammar [5], but can be adequately implemented by expansions using type and relation definitions.

As a first example, consider the canonical graph (37) for the verb "connect." This graph can join Graphs (10) and (11) of the block diagram basis only on the **device** concepts. But, there is little motivation to arbitrarily attach a port or carrier concept to a device. The verb "connect," however, clearly implies the existence of a communicating carrier between the two devices, and, furthermore, the preposition "to" suggests at least partial specification of directionality. In comparison, the phrase "with device" suggests no directionality, so the (<>) relation is required. Consider the schema for **connect** shown in Graph (42). While this graph could be used as a canonical graph for "connect," it should not be the first choice, since an incorrect attachment during semantic analysis based on the implied carrier might be made over a correct attachment to the object device. For the **connect** concept to occur in a conceptual graph, an instantiation of the canonical graph for the word "connect" or a synonym will be a subgraph of it. First, this schema can be joined with the canonical graph for connect in the given sentence graph. Next, the subgraph of this schema that identifies the ports and their attachment relations is projected into the given block diagram graph. The two graphs may then be joined on the projection. (If multiple projections occur, the schema must be retracted.)

schema for connect(x) **is**

```
[connect : *x] -
   (inst) -> [carrier] -
                  (<>) -> [port: *1]
                  (<>) -> [port: *2],
   (obj) -> [device] -> (->) -> [port: *1]
   (dest : {to}) -> [device] -> (<-) -> [port: *2],.        (42)
```

A more subtle implication to facilitate the joining of structural information from block diagrams with natural language statements involves the **send** concept. Sending a signal or message from one device to another implies a partially directed connection between the two, so the canonical graph (43) below for "send" may be expanded by schema (44). This concludes that a carrier exists, if one is not specified by a "using carrier" or a "on carrier" phrase, and that the agent device (subject of an active sentence) is the source and the device introduced by the preposition "to" is the destination.

```
[send] -
        (agnt) ->[device]
        (obj) ->[value]
        (dest: {to}) ->[device]
        (inst: {using | on}) ->[carrier]                  (43)
```

schema for send(x) **is**

```
[send : *x] -
   (agnt) ->[device] -> (->) ->[port: *1]
   (obj) ->[value]
   (dest: {to}) ->[device] ->(<-) ->[port: *2]
   (inst: {using | on}) -> [carrier] -
                                (<>) ->[port: *1]
                                (<>) ->[port: *2]    (44)
```

As a third example, consider that Graph (38) for "consist of" does not integrate well with the structural basis Graph (12). The problem is the separation of the containing device and the contained devices by the "contain" node and its attached relations. This can be improved by (45), where **part** relations are added from the agent device of **contain** to each component device of its object.

In addition to the structural implications of some canonical graphs for natural language, causal and temporal relations may also be implied for integration with graphs produced from flowcharts and timing diagrams. For example, graphs such as (39) containing conditional expressions (**if** relations) need to be augmented by the starts_when_finished_if (**swfif**) relation used in the flowchart canonical basis. This

can be done with a relation definition corresponding to the **if** relation in English statements.

schema for contain(x) **is**

```
[contain : *x] -
    (agnt) -> [device] -
        (part) -> [device: *a]
        (part) -> [device: *b]
        (part) -> [device: *c],
    (obj) -> [device : AND] -
        (part) -> [device: *a]
        (part) -> [device: *b]
        (part) -> [device: *c],,.        (45)
```

schema for if(x, y) **is**

```
[action: *x]
        (if) -> [condition: *y]
        (swsif) -
                1 -> [action: *x]
                2 -> [condition : *y],,.  (46)
```

Similarly, an implied starts_when_finished relation can be added to a cause concept.

schema for cause(x) **is**

```
[cause : *x] -
    (agnt) -> [action: *x]
    (obj) -> [action] -> (swf) -> [action: *x],.   (47)
```

The causal relation implied by using one action as the agent of another in an English expression can be expanded to include an **swf** relation with a relation definition (48) for the agent relation.

schema for agnt(x,y) **is**

```
[action: *x] -
        (agnt) -> [action: *y]
        (swf) -> [action: *y]            (48)
```

A very important aspect of the integrated conceptual graph that is formed by these and other integration mechanisms is that it be self-consistent. In the next section, analysis of conceptual graphs is considered.

5 Analysis

The conceptual graph formed by the integration of individual requirements graphs can be subjected to several kinds of analysis. First, consistency can be checked by determining if any pair of concept nodes have inconsistent relations between them. Second, translation and integration employ canonical basis graphs and schemata which introduce new generic concepts. Since a generic concept is a concept which has no referent, generic concepts may be interpreted as omissions, particularly generic concepts having few relations with other concepts of the integrated graph. Additional analyses of integrated graphs for measures such as cost, performance, and reliability can be considered, but are beyond the scope of this discussion.

5.1 Consistency

Assuming the conceptual bases used to form the conceptual graphs are self-consistent and their concepts (and relations) have been restricted with consistent (conformal) subtypes and individuals, the primary manifestation of inconsistency will be through inconsistent conceptual relations between a given set of concepts, or between sets of concepts tentatively joined by the **same** relation. In examples studied thus far, conceptual graphs tend to be relatively sparse, having few relations between pairs (or sets) of concepts. This suggests the computational cost of consistency checking will be small, particularly since conceptual relations are typed, and many types of relations are simply not comparable, such as **obj** with **if**. On the other hand, a relation between two concepts may be inconsistent with the transitive closure (composition) of relations along some other path between those concept nodes. Detecting this form of inconsistency globally requires the additional computation of forming the transitive closure and comparing the new relations generated. Fortunately, many relations do not have transitive properties. For example, the composition of **agnt** with itself is meaningless. Also, the composition of relations can be computed as graphs are joined, distributing the computation over time. In the following, it will be assumed that the transitive closure has been performed. Developing consistency / reduction tables for all pairs of relations and estimating the computational cost of consistency checking are topics of current research.

The checking of pairs of relations between given concepts is facilitated by consistency tables for appropriate types (and subtypes) of relations. In some cases, a pair of relations may not only be consistent, but may be replaced by a single relation. Thus, consistency checking can provide an added benefit of reducing the conceptual graph.

As a first example, a consistency / reduction table for interconnection relations in block diagrams can be constructed as shown in Table 8 for block diagram attachment relations. Two interconnection relations between a pair of concepts either reduce or conflict as shown in the table. Interconnection inconsistencies are denoted by an 'X' in the table. For example, attachment between a pair on nodes cannot be both **input_only** (<<) and **output_only** (>>), but an input attachment (<-) and an output attachment (->) between the same pair of nodes is simply a bidirectional attachment (<>).

TABLE 8
CONSISTENCY/REDUCTION TABLE
FOR INTERCONNECTION RELATIONS

	--	->	<-	>>	<<	<>
--	--	->	<-	>>	<<	<>
->	->	->	<>	>>	X	<>
<-	<-	<>	<-	X	<<	<>
>>	>>	>>	X	>>	X	X
<<	<<	X	<<	X	<<	X
<>	<>	<>	<>	X	X	<>

Two **part** relations between a pair of devices must either simplify to one part relation or must conflict (if they are in opposite directions). Since the **part** relations are composi-

tional, the transitive closure of part relations should be computed and checked.

Since the relations used in flowchart canonical graphs are very like those used in for timing diagrams, they may be considered together. For example, the **swfif** relation can be considered to be a **swf** relation for consistency analysis, but not for reduction. Since the composition of temporal relations is possible, the transitive closure should be performed before the consistency check is made. In comparing temporal relations, the inverse relations must also be considered. The finishes_before_starts (fbs) and its inverse must also be included in the analysis. This results in a 12 by 12 table, which may be found in Matuszek et al. [20].

In addition to the **swf** and **swfif** relations in flowcharts, the unary **entry** and **exit** relations may also occur. Since these are interruptions on flowlines, and flowlines are unidirectional, a process may have multiple entry or multiple exit relations, but each must have a distinct referent.

The relations used in the natural language canon may be partitioned according to whether their initial concept nodes may be **actions** and **events**, or **devices** and **values**. Relations from actions and events include the case frame relations: **accompaniment**, **agent**, **destination**, **instrument**, **manner**, **object**, **purpose**, and **source**. Because of the wide variety of constructs that may occur in English it is difficult to determine which pairs of relations are universally inconsistent. But for the sublanguage considered in Table 9 is useful. Incompatible pairs of relations are marked with an "X." Relation pairs marked by a question mark are generally not compatible, but may occur in rare cases with a reflexive pronoun as in "the processor (agnt) sent itself (dest) a message," "the device (agnt) started itself (obj)," "the device added the value (obj) with itself (acc)."

TABLE 9
CONSISTENCY FOR NATURAL LANGUAGE RELATIONS

	acc	agnt	dest	inst	man	obj	purp	src
acc		X	X	X	X	?	X	X
agnt	X		?	X	X	?	X	?
dest	X	?		X	X	X	X	?
inst	X	X	X		X	X	X	X
man	X	X	X	X		X	X	X
obj	?	?	X	X	X		X	?
purp	X	X	X	X	X	X		X
src	X	?	?	X	X	?	X	

The relations **ordinal**, **quantity**, and **attribute** which attach to **device** and **value** nodes are mutually inconsistent since ordinals, quantities, and attributes point to different concept types. Another type of inconsistency can be checked for with the ordinal and quantity relaions. An object cannot be related to two different ordinals (as the second and the seventh register) and cannot be two different quantities, but may have several different attributes.

Finally, the absurd relation and relations with absurd and empty concepts need to be mentioned. Absurd relations and concepts and empty concepts can only arise through canonical graphs and type definitions. The negative constraints they impose can also be represented through inconsistency tables and "false" graphs, though this may result in a higher computational cost. The absurd relation is interpreted as a statement that no relation can exist between the concepts in question, so any other relation between the concepts resulting from a join must be inconsistent. (Similarly, the universal relation between two concepts mean that they are identical and should be joined.) A relation with the absurd concept means that that particular relation cannot exist for any possible concept, and so that particular relation from the same initial concept to any other concept must be an error. A relation with an empty concept of a particular type indicates that that relation cannot exist with an individual of that type. Any join that contradicts this must be an error.

As research continues, a more complete basis for consistency checking will be developed. Next, the analysis of requirements for omissions is considered briefly.

5.2 Completeness

Determining what is missing in a requirements document is normally a difficult task, but is relatively simple using conceptual graphs. The canonical graphs forming the canonical bases for translating source notations are templates representing the expected context or environment of a concept. Similarly, schemata are templates. A simple indication of completeness in a template system is the presence of unfilled template slots. In basis graphs and schemata, all nodes are initially generic (unfilled). As a requirements document is completed, many of the generic concepts should become individualized or should become specialized by many relations including same relations becoming attached to them. Concepts that are not specialized and remain only loosely connected to the the integrated graph may represent omissions in the requirements document. (This is not to say that they must be specified in this document, but that task may be delegated to designers.) The value of this form of omission analysis is that the requirements authors can be made aware of the decisions that are being passed on to the design engineers. Alternatively, loosely coupled nodes may represent irrelevancies, and should be deleted by the requirements author. For example, the extent of specialization of a node is represented by the degree of the node (especially the out-degree). Detecting low-degree nodes with respect to a histogram of node degrees should quickly identify problems.

An integrated requirements graph should be roughly centered about a concept node labeled by the system type. If requirements are well balanced, new concepts introduced are later specialized by further statements and expressions, and the graph should grow relatively evenly from the initial system concept node. A graph which has grown extensively along only a few paths suggests that some aspects of the system have been overspecified, while other aspects are incomplete.

These are only a couple of example of how a requirements graph may be analyzed for completeness (and overspecification). Further algorithms to determine the quality of a specification based on analysis of the graph offer fertile ground for research.

6 FUTURE RESEARCH

This work will be extended in several ways in future research. First, conceptual bases need to be developed for additional source notations employed in requirements documents. In particular, the flowchart notation does not admit multiprocessing and parallel processing, so a notation such as Petri nets will be considered included. Another notation that may be observed in product specifications is the state diagram (often as an informal bubble diagram). the hierarchical organization of states in statecharts will also be accommodated Finally, tables are used quite extensively in requirements documents to specify such diverse requirements as the results of various signal values on ports and ranges of values for delays and other parameters. Although a table may be construed as a small relational data base, generating conceptual graphs that integrate easily with other requirements should be an interesting task.

The current canonical basis for English is quite limited, and must be extended, especially in type definitions. Much more experience with the analysis of requirements expressed in natural language is needed to provide a robust and user friendly system.

An exciting direction for future research is automatic synthesis of systems at the conceptual graph level, that is requirements-level synthesis. To some degree this involves developing algorithms to perform traditional synthesis tasks (design style selection, partitioning, scheduling, allocation) on conceptual graph representations. But, a set of requirements is not a design, and therefore will provide an incomplete model at best. To overcome this, a prototype can be developed for each concept in order to provides a typical or normal context or instance of that concept by including default values of measurement concepts and other reasonable assumptions. The result of joining such default prototypes to a conceptual graph of requirements is a preliminary design. This design may then be analyzed for cost and performance, and may even be complete enough to support some form of simulation. Initial progress in automatically generating graphical feedback models and in synthesizing VHDL descriptions from conceptual graphs are encouraging [15], [30].

ACKNOWLEDGMENTS

The author would like to acknowledge that the research reported here has been shaped and spurred on by a large number of colleagues and friends over the years. In particular, the author would like to thank John Sowa, Jim Armstrong, Rob Greenwood, Aniruddha Thakar, and Shawn Neugebauer for their very helpful conversations and communications.

This work was supported, in part, by Semiconductor Research Corporation Contract 92-DJ-230, Virginia's Center for Innovative Technology Grant INF-92-005, and National Science Foundation Grant MIP-9120620.

REFERENCES

[1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling, Vol. 1: Parsing*, Prentice Hall, Englewood Cliffs, N.J., 1972.

[2] J.F. Allen, "Towards a General Theory of Action and Time," *Artificial Intelligence*, vol. 23, no. 2, pp. 124-154, July 1984.

[3] D. Burnette and J. Armstrong, "Automated Assists to the Behavioral Modeling Process," *Proc. First Int'l Workshop Rapid System Prototyping*, pp. 187-195, 1991.

[4] R.R. Burton, *Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems*, BBN Report #3453, Bolt, Beranek, and Newman, Cambridge, Mass, Dec., 1976.

[5] *Graph Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig, and G, Rosenberg, eds., Springer-Verlag, Berlin, 1979.

[6] W.R. Cyre, "Toward Synthesis from English Descriptions," *Proc. 26th Design Automation Conf.*, Las Vegas, Nev., pp. 742-745, June 1989.

[7] W.R. Cyre, "Mapping Design Knowledge from Multiple Representations," *Proc. 1991 IEEE Int'l Conf. Computer Design (ICCD '91)*, Boston, Oct. 1991.

[8] W.R. Cyre, "The Conceptual Representation of Waveforms for Temporal Reasoning," *IEEE Trans. Computers*, vol. 43, no. 2, pp. 186-200, Feb. 1994.

[9] H.S. Delugach, "Specifying Multiple-Viewed Software Requirements with Conceptual Graphs," *J. Systems Software*, vol. 19, pp. 207-224, 1992.

[10] D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *Proc. Int'l Conf. Software Eng.*, Apr. 1988.

[11] J. Esch and T.E. Nagle, "Representing Temporal Intervals Using Conceptual Graphs," *Proc. Fifth Workshop Conceptual Structures*, Boston, July 1990.

[12] C.J. Fillmore, "The Case for CASE," *Universals in Linguistic Theory*, Bach and Harms, eds., Holt, Reinhart and Winston, Chicago, pp. 1-90, 1968.

[13] J.J. Granacki and A.C. Parker, "PHRAN-SPAN: A Natural Language Interface for System Specifications," *Proc. 24th Design Automation Conf.*, June 1987.

[14] R. Hartley and M. Coombs, "Reasoning with Graph Operations," J. Sowa, ed., *Principles of Semantic Networks*, Morgan Kaufmann, San Mateo, Calif., 1991.

[15] A. Honcharik, "Generation of VHDL from Conceptual Graphs of Informal Specifications," MS thesis, Virginia Tech, 127 pages, Aug. 1993.

[16] "IEEE Standard VHDL Reference Manual," IEEE, New York, 1988.

[17] "ExpressV-HDL User and Reference Manual," i-Logic Inc., Burlington, Mass., Jan. 1992.

[18] *Sublanguage: Studies of Language in Restricted Semantic Domains*, R. Kittredge and J. Lehrberger, eds., deGruyter, New York, 1982.

[19] D.W. Knapp and A.C. Parker, "A Unified Representation for Design Automation," C.J. Koomen and T. Moto-oka, eds., *Computer Hardware Description Languages and Their Applications*, Tokyo, pp. 29-31, Aug. 1985.

[20] D. Matuszek, T. Finin, R. Fritzson, and C. Overton, "Endpoint Relations on Temporal Intervals," *Proc. Third Ann. Rocky Mountain Conf. Artificial Intelligence*, pp. 182-188, June 1988.

[21] M. Mugnier and M. Chein, "Polynomial Algorithms for Projection and Matching," *Proc. Seventh Ann. Workshop Conceptual Structures*, Las Cruces, N.M., pp. 49-58, July 8-10, 1992.

[22] C.S. Peirce, *Collected Papers of Charles Sanders Peirce*, A.W. Burks, ed., eight volumes, Harvard Univ. Press, Cambridge, Mass., 1960.

[23] J.L. Peterson, "Petri Nets," *Computing Surveys*, Sept. 1977.

[24] E. Rich and K. Knight, *Artificial Intelligence*, McGraw-Hill, New York, 1991.

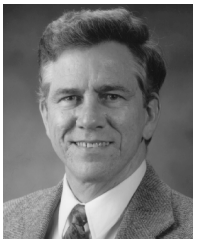
[25] R.C. Shank and B.L Nash-Webber, *Theoretical Issues in Natural Language Processing*, Assoc. of Computational Linguistics, 1975.

[26] S. Shankaranarayanan and W.R. Cyre, "Identification of Coreferences with Conceptual Graphs," *Proc. 1994 Int'l Conf. Conceptual Structures*, College Park, Md., pp. 45-60, Aug. 16-20, 1994.

[27] J.F. Sowa and E.C. Way, "Implementing a Semantic Interpreter Using Conceptual Graphs," *IBM J. Research and Development*, vol. 30, pp. 57-69, Jan., 1986.

[28] J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, Mass., 1984.

[29] *EDIF Electronic Design Interchange Format Version 2.0.0*, P. Stanford and P. Mancuso, eds., Electronics Industries Assoc., 1989.
[30] A. Thakar and W. Cyre, "Visual Feedback for Validation of Informal Specifications," *Proc. Int'l Workshop Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '94)*, Durham, N.C., pp. 411-412, Jan. 31-Feb. 2, 1994.
[31] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: A Language for System Level Synthesis," *Proc. Computer Hardware Description Languages*, pp. 145-154, 1991.
[32] T. Winograd, *Language as a Cognitive Process, Vol. 1: Syntax*, Addison-Wesley, Reading, Mass., 1983.



Walling R. Cyre (S'69-M'73) received the BS, MS, and PhD degrees in electrical engineering from the University of Florida in 1965, 1970, and 1973, respectively. In 1974, he joined the faculty of the University of Wisconsin at Madison and, in 1977, he joined Control Data Corporation in Minneapolis to design signal processing systems for real-time applications. During his last five years with Control Data Corporation, Dr. Cyre served as an internal technical consultant in electrical computer-aided design research group and as an industrial mentor of university research projects in computer-aided design.

Currently, Dr. Cyre is an associate professor of electrical engineering at Virginia Polytechnic Institute and State University (Virginia Tech), where he teaches courses in system-level design automation, computer architecture, microprocessor systems design, hardware design, and electrical circuits. His research interests include the application of artificial intelligence to automated design and synthesis from informal specifications, conceptual modeling of computing systems and timing analysis. He is a member of the IEEE and the IEEE Computer Society.