Task-Based Specifications Through Conceptual Graphs

Jonathan Lee, Lein F. Lai, and Wei T. Huang, National Central University

Conceptual modeling is an important step toward the construction of user requirements. Requirements engineering is knowledge-intensive and cannot be dealt with using only a few general principles. Therefore, a conceptual model is domain-oriented and should represent the richer semantics of the problem domain. The conceptual model also helps designers communicate among themselves and with users.

To capture and represent a conceptual model for the problem domain, we need

- mechanisms to structure the knowledge of the problem domain at the conceptual level, which has the underlying principles of abstraction and encapsulation; and
- formalisms to represent the semantics of the problem domain and to provide a reasoning capability for verification and validation.

We propose the task-based specification methodology as the mechanism to structure the knowledge captured in conceptual models. TBSM offers four main benefits for constructing conceptual models:

First, incorporating the task structure provides a detailed functional-decomposition technique for organizing and refining functional and behavioral specifications.

Second, the distinction between soft and rigid conditions lets us specify conflicting functional requirements.

Third, with TBSM, not only can we document the expected control flow and module interactions; we can also verify that the behavioral specification is consistent with the system’s functional specification.

Fourth, the state model makes it easier to describe complex state conditions. Terminology defined in the state model can easily be reused for specifying the functionality of different tasks. Without such a state model, describing the state conditions before and after a functional unit of an expert system is too cumbersome to be practical.

We propose conceptual graphs as the formalism to express task-based specifications where the task structure of problem-solving knowledge drives the specification, the pieces of the specification can be iteratively refined, and verification can be performed for a single layer or between layers. We chose conceptual graphs for their expressive power to represent both declarative and procedural knowledge, and for their assimilation capability—that is, their ability to be combined.

Structuring knowledge as tasks

TBSM acquires and organizes domain knowledge, functional requirements, and high-level problem-solving methods. A spec-
ification has two components: a model specification and a process specification.

**Model specification.** The model specification describes the system’s static properties, and consists of two models: the domain model, a model about domain objects, and the state model, a model about the problem-solving states.

The domain model is similar to an entity-relationship model—that is, the domain model captures terms, attributes, relations, and cardinality. A constraint in TBSTM is characterized by parts, facets, and rules. Parts include a constraint’s inputs and outputs. There are four facets: scope, strength, mapping, and property. A constraint’s scope can be global or local; its strength can range from required (strongest, denoted as $C_0$) to preference (weak, denoted as $C$); the mapping is multivalued or single-valued; and its property indicates whether it is positive or negative.

The state model defines the terms (called state objects) that describe problem-solving states. In TBSTM, a state object can describe what problem-solving stage the task is at (the stage of completion), as well as dynamic relationships among multiple objects (constraints satisfaction).

**Process specification.** The process specification characterizes the system’s dynamic properties, using state transitions and task state expressions (TSEs).

To explicitly specify what a task is, we treat a task as a state transition. To specify the states before, during, and after the operation, we use preconditions, postconditions, and protections, which are partial descriptions of the state. A task’s precondition describes the situation under which the task can be invoked. A postcondition describes desirable state changes that the task should achieve, and is either rigid or soft. A rigid postcondition must always be satisfied; a soft postcondition can be satisfied to a degree. A protection limits coupling between modules (tasks).

TSEs specify a target system’s behavior. A TSE defines the desirable sequencing of tasks that should be processed before the given task, the interaction of tasks at different levels, and disallowed sequencing (such as a TSE is preceded by a negation, denoted as $\neg$).

A TSE uses these operators:

- **Immediately follow** (concatenation): indicated by a juxtaposition of tasks, separated by a comma. That is, a task immediately follows the previous one.
- **Follow** represented by a semicolon. For instance, $(T_1; T_2)$ specifies that $T_2$ follows $T_1$, with a sequence of intermediate (unspecified) tasks in between (that is, $T_1, \ldots , T_2$).
- **Selection** (between multiple control flows): represented by $(e_1 \land e_2)$, where $e_1$ and $e_2$ are two TSEs that represent two mutually exclusive control flows.
- **Optional**: represented by $[e]$, which is a shorthand notation for $(e \lor \neg e)$, where $e$ is the null operation.
- **Conditional**: indicated by a logic formula $\beta$ that is attached to a TSE $e$ to express that $e$ will be invoked if $\beta$ is true. For instance, if $\beta_1$ then $e_1$, else if $\beta_2$ then $e_2$, ... ; else if $\beta_n$ then $e_n$, denoted as $e_1 \land (\beta_1 \lor e_2 \land (\beta_2 \lor \ldots \lor (e_n \land \beta_n)))$.

We chose conceptual graphs for their expressive power to represent both declarative and procedural knowledge, and for their ability to be combined.

$e_1, \ldots , e_n$; else if $\beta_n$ then $e_n$ can be represented as $\beta_1 e_1 \lor \beta_2 e_2 \lor \ldots \lor \beta_n e_n$.

**Iteration**: represented by $(\beta \mathcal{E})$, where $\beta$ is an iteration condition.

These operators fall into three groups: sequencing (follow and immediately follow), branching (selection, optional, and conditional), and iteration. Operators in the same group, but with different degrees of specificity, complement the refinement process used during requirements analysis. Initially, we can use less specific operators, such as follow for the sequencing, selection and optional for the branching, and iteration without an iteration condition. After the refinement, as more information becomes available, we can use more specific operators (for example, immediately follow and conditional) to describe further details of the specifications.

A TSE can be at the task or method level. A task-level TSE associated with a task $T$, denoted as $\varepsilon_T$, documents global interactions between the current task and tasks at different levels. Task-level TSEs often use follow to describe partial task sequences. A method-level TSE documents the local control flow among the method’s subtasks but typically does not contain the follow operator.

Multiple TSEs must often be combined to obtain a global view of the system’s behavior from pieces of local behavior specifications. To achieve this, we develop a composition operator for a task $T$, denoted as $\mathcal{E}_T$, which combines $T$’s TSE, denoted as $\varepsilon_T$, and the TSE of $T$’s parent method, denoted as $\varepsilon_m$. The composition operator synthesizes $\varepsilon_T$ with the components of $\varepsilon_m$ in which $T$ is always invoked. Some general cases of the composition results are:

- $\varepsilon \mathcal{E}_T \varepsilon_T = \varepsilon \lor \varepsilon_T$, if $\varepsilon$ does not contain $T$
- $(\alpha, T, \beta, \gamma_1, \gamma_2) \mathcal{E}_T (T, \beta, \gamma_1, \omega) = (\alpha, T, \beta, \gamma_1, \gamma_2) \lor (\varepsilon_T \mathcal{E}_T \varepsilon_T)

Refining and verifying specifications. TBSTM supports the refinement of the model and the process specifications. Both specifications can be first described in their high-level abstract forms, which can be further refined into more detailed specifications in the next level. We adopt the notion of task structure (that is, task/method/subtask) for the process refinement. That is, refining a process involves refining a task into a set of problem-solving methods or domain-specific methods that can accomplish the task, and refining a method by specifying subtasks and then using TSEs to describe temporal relationships among them.

TBSTM also performs verification for consistency and completeness for pieces of the specification in one abstraction level or between multiple levels, based on the formal semantics of TBSTM.

Using TBSTM for R1/SOAR. To illustrate the various components in task-based specifications, let’s examine an example based on a subset of the TBSTM specification of R1/SOAR. R1/SOAR is an expert system that focuses on the unix configuration portion of R1, an expert system for computer configuration. We’ll focus on two tasks: Configure-Modules and Configure-a-Module. Configure-Modules configures as many modules as possible into
the current backplane while maintaining the optimal ordering of modules. The task achieves this by iterating between two subtasks: Configure-a-Module, which attempts to configure the current module in the current backplane, and Obtain-the-Next-Module, which gets an ordered module that follows the current module in the optimal ordering.

A correct specification should indicate that Obtain-the-Next-Module should be skipped if the current module fails to be configured into the current backplane, because we have not finished configuring the current module. The specification should also indicate that, if the current module can not be configured into the current backplane, the system should later find a backplane suitable for this module before configuring the next module.

Figure 1 shows the specifications for the tasks and methods relevant to this example.

Expressing task-based specifications in conceptual graphs

The conceptual graph is a directed, finite, connected graph that consists of concepts, concept instances (referents), and conceptual relations. Concepts and relations represent declarative knowledge. Procedural knowledge can be attached through actors, which represent processes that can change the referents of their output concepts, based on their input concepts. A type lattice can specify the taxonomy of concepts, relations, and actors. Concepts are enclosed by square brackets, relations by parentheses, and actors by angle brackets. A relation node contains a type label only. A concept node contains a type label and a referent field. The referent field can be an individual marker (a unique individual identifier), a constant (a number or string), a variable, sets, or the generic marker (*). Table 1 summarizes different types of referents.

Each concept and relation type has a canonical graph and a type definition. The canonical graph specifies the selection constraints for that type. The type definition represents a type's necessary and sufficient conditions. A context is represented by a concept graph whose referent is a set of conceptual graphs that are being asserted. A schema specifies the expected defaults and other background knowledge. We can derive new conceptual graphs from old graphs by using graph-formation rules, as we'll show later.

Harry S. Delugach extended conceptual graphs to include demons, a type of node (in double-angle brackets) that creates and retracts input and output concepts. A conceptual graph that uses demons is a temporal conceptual graph. A demon’s algorithm asserts (that is, marks) each of the demon’s
Table 1. Types of referents. \( T \) denotes the type of a concept, \( S \) denotes a set, and \#\( S \) denotes a designated set.

<table>
<thead>
<tr>
<th>Referent Type</th>
<th>Expression</th>
<th>Semantics</th>
<th>Example</th>
<th>English expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>([T:*])</td>
<td>(\exists x, x \in T)</td>
<td>([Cat:*])</td>
<td>A cat or some cat</td>
</tr>
<tr>
<td>Individual</td>
<td>([T:#n])</td>
<td>(#n \in T)</td>
<td>([Cat:#02])</td>
<td>The cat #02</td>
</tr>
<tr>
<td>Proper name</td>
<td>([T:Name])</td>
<td>Name ( \in T)</td>
<td>([Cat:Kitty])</td>
<td>The cat named Kitty</td>
</tr>
<tr>
<td>Unique</td>
<td>([T:#1])</td>
<td>(\exists ! x, x \in T)</td>
<td>([Cat:#1])</td>
<td>One and only one cat Kitty and Garfield Cats or some cats</td>
</tr>
<tr>
<td>Set</td>
<td>([T:({N_1, \ldots, N_k})])</td>
<td>(N_1, \ldots, N_k \subseteq T)</td>
<td>([Cat:({Kitty, Garfield})])</td>
<td>Four cats</td>
</tr>
<tr>
<td>Plural set</td>
<td>([T:(*)])</td>
<td>(\exists S, S \subseteq T)</td>
<td>([Cat:(*04)])</td>
<td>The cats Every cat</td>
</tr>
<tr>
<td>Definite set</td>
<td>([T:#(\ast)])</td>
<td>(#S \subseteq T)</td>
<td>([Cat:#(\ast)])</td>
<td>No cat</td>
</tr>
<tr>
<td>Universal</td>
<td>([T:V])</td>
<td>(\forall x, x \in T)</td>
<td>([Cat:V])</td>
<td></td>
</tr>
<tr>
<td>Negative</td>
<td>([T:-])</td>
<td>(\forall x, x \notin T)</td>
<td>([Cat:-])</td>
<td></td>
</tr>
</tbody>
</table>

output concepts with referents, and retracts each of the demon’s input concepts. If more than one input concept exists, no demon action occurs until all of the demon’s input concepts have been asserted. Each demon possesses a canonical definition describing what the demon’s input concepts are and what its output concepts will be.

The conceptual graphs also incorporate constraint overlays, which allow the attachment of constraints to objects and collections of objects that make up world states. Constraint overlays, represented in angle brackets and linked to concepts with dashed lines, can overlay actors (procedures or constraints) on a conceptual graph to describe the changes to a model state.

Figure 2 provides an overview of the mapping from TBSM to conceptual graphs. We’ll explain this mapping in more detail in the following sections.

Issues in mapping task-based specifications to conceptual graphs. Conceptual graphs have several features that can facilitate the mapping from task-based specifications to their counterpart graphs. For example, terms and relations in the domain model of TBSM can be directly mapped to concepts and conceptual relations, and the notion of partial conceptual graphs corresponds to the notion of partial models. However, using conceptual graphs to represent task-based specifications raises several issues:

First, we use conceptual graphs to represent metalevel information such as tasks and methods. The general notion of “task” is a key element of our methodology. Without this notion, we cannot describe task decomposition and various temporal relationships among tasks, let alone combine the partial process specifications.

Second, a constraint is defined in domain models, and used in state models. Therefore, a constraint representation in conceptual graphs should focus on not only the declarative aspect, but also the procedural.

Third, a state can be changed either by the performance of a task or through a constraint satisfaction. A conceptual graph representation of a state model should be able to capture the semantics for the state change.

Finally, because TSEs are an extension of regular expressions, they can be represented by state-transition diagrams. However, it is hard to represent states because they are implicit in TSEs. We must also deal with the implementation of the composition operator in conceptual graphs.

Model specifications. We use two conceptual relations, (domain-model) and (state-model), to describe the conceptual graph representation of a task’s model specification. (domain-model) is defined by a domain model concept that connects to a concept of type TASK and a concept of type GRAPH using the generic relation (link) (see Figure 3). The concept [Domain-Model] describes domains of a software model, and is a part of the static properties that are attributes of the world being modeled (see Figure 3). As the figure shows, the definitions of (state-model) and (domain-model) are similar. The concept [State-Model] describes states of a software model, and is part of the static properties that are attributes of the world being modeled.

Domain model. Because models are similar to entity-relationship models, the transformation of terms, attributes, relations, and
cardinality to conceptual graphs is straightforward. A term becomes a concept of type ENTITY. A relation (attr) represents an attribute associated with a term, and a relation (chr) shows that a data type (which is a concept type) characterizes an attribute name. A relation becomes a conceptual relation. To convert the cardinality of each referent of a concept, we adopt these rules:

- One-to-one: \([E_1;(^+\circ\@1]} \rightarrow (R) \rightarrow [E_2;(^\circ\@1]}]
- One-to-many (or many-to-one): \([E_1;(^+\circ\@1]} \rightarrow (R) \rightarrow [E_2;(^\circ\@n}]
- Many-to-many: \([E_1;(^\circ\@n}] \rightarrow (R) \rightarrow [E_2;(^\circ\@m}]
- One-to-many, one-to-many: \([E_1;(^x\circ\@1}] \rightarrow (R_1) \rightarrow [I \rightarrow [E_2;(^y\circ\@1}] \rightarrow (R_2) \rightarrow [E_2;(^x\circ\@n}] \rightarrow [E_2;(^y\circ\@m}]

To represent a TBSM constraint in a conceptual graph, we must translate its parts, facets, and rules. Parts are inputs and outputs to a constraint. The facets—scope, strength, mapping, and property—are attributes associated with the constraint in its counterpart graph. The relation (accm), which means accompaniment, connects a constraint and its rule, which consists of a rule name and a rule body. We implement the rule body as an actor. A fix that repairs a constraint violation also translates into an actor. Figure 4 describes the general form for a constraint representation in conceptual graphs.

So, we can establish a constraint hierarchy based on each constraint’s strength. The constraint hierarchy is useful for reasoning about constraints, using constraint satisfaction and relaxation. Consider the task specification of Configure-a-Module in Figure 1c. Figure 5 shows the specification of the domain model of this task transformed into its counterpart conceptual graph. One of the constraints in the task is also transferred into its conceptual graph.
State model. A conceptual graph representation for a state model should capture two main semantics in the model: the stages of completion and constraints satisfaction. We follow two basic translation rules. First, we overlay constraint actors on the conceptual graph of state objects. The constraint overlays show the relationships among state objects. All state objects associated with a constraint actor must satisfy the relationship expressed by the actor. Second, we use demons for state transitions whose inputs and outputs are state concepts as referents of the type STATE, and whose input tokens are of the type TASK. No demon action occurs until all input concepts are asserted and all input tokens are enabled. The performance of tasks (in the form of input tokens) is thus essential for state transitions. Figure 6 shows the canonical graph of demons, and Figure 7 shows the conceptual graph for the state model of Configure-a-Module.

Process specifications. We’ll now show how we translate the TBSM’s functional and behavioral specifications to a conceptual graph.

Functional specifications. Propositions represent preconditions, protections, and rigid postconditions. A soft postcondition can be viewed as a fuzzy proposition with fuzzy concepts and fuzzy conceptual relations. Figure 8 specifies the conceptual graph representation of the functional specification of Configure-a-Module. A disjunctive postcondition is transferred into an equivalent proposition—that is, \( p \lor q = \neg (\neg p \land \neg q) \).

Behavioral specifications. A \((\text{task})\) relation associates a TSE with its task in conceptual graphs. Figure 9 shows the relation’s definition. The concept \([\text{Behavior-Model}]\) describes a software model, and is a part of the dynamic properties that can be considered as attributes of the world being modeled. Figure 9 shows the type definition for the behavior model.

As we mentioned before, the TSE is an extension of regular expressions and therefore can be represented by state-transition diagrams. To transform TSEs into conceptual graphs, we use demons to represent transitions. A demon possesses the semantics of an actor node with respect to output concepts’ referents, with the additional semantics that the demon asserts its output concepts and then retracts its input concepts. We also assume that a mapping \( g \) exists that maps an expression to its state where the postcondition, after processing through the expression, is true.

To distinguish between the follow and immediately follow operators, we use a demon for immediately follow and mark the input tasks, whereas we transform follow into a demon whose inputs are not yet completely marked.

We adopt two conventions for the selection, iteration, conditional, and optional operators. First, we use an initiator demon (that is, \(<\text{CT}>\)) and a START concept (a subtype of \([\text{STATE}]\)) for the beginning state. Two graphs cannot join on \([\text{START}]\) because it is an unknown state, unless the same event invokes the transition that follows. Second, we view a conditional test as a task, denoted as \( \beta \), where \( \beta \) is a condition. That is, \( \beta \) is a special control-flow task that is invoked only when \( \beta \) tests to be true, whereas \( \neg \beta \) is a special control-flow task that is invoked only when \( \neg \beta \) tests to be true. We denote a final state by attaching the monadic relation (final).
The difference between selection and conditional operators is that, in the conditional operator, the expression \( e \) will not be performed unless the conditional test for \( \beta \) tests to be true, and neither \( e \) nor \( \beta \) is marked. We treat an optional operator as a special case of a selection operator. Three demons implement an iteration operator. The first demon is invoked by performing an expression \( e \), while the second demon will be invoked by two input tokens, 'opt' and 'e', to represent the iteration condition. The third demon indicates the exit condition. Table 2 summarizes the mapping of TSE operators to conceptual graphs. Figure 10 describes the conceptual graph specification of a TSE for the task Configure-a-Module.

Table 2. Mapping TSE operators to conceptual graphs.

<table>
<thead>
<tr>
<th>TSE OPERATORS</th>
<th>SYNTAX</th>
<th>IN CONCEPTUAL GRAPHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediately follow ((T_1, T_2))</td>
<td>(\longrightarrow \text{&lt;transition&gt;} \longrightarrow [\varphi(T_1)] \longrightarrow \text{&lt;transition&gt;} \longrightarrow [\varphi(T_2)])</td>
<td>[(T_1)] [(T_2)]</td>
</tr>
<tr>
<td>Follow ((T_1, T_2))</td>
<td>(\longrightarrow \text{&lt;transition&gt;} \longrightarrow [\varphi(T_1)] \longrightarrow \text{&lt;transition&gt;} \longrightarrow [\varphi(T_2)])</td>
<td>[(T_1)] [(T_2)]</td>
</tr>
<tr>
<td>Selection ((e_1 \lor e_2))</td>
<td>(\text{&lt;T&gt;} \longrightarrow \text{&lt;START&gt;} \longrightarrow \text{&lt;transition&gt;} \longrightarrow [\varphi(e_1)])</td>
<td>[(e_1)]</td>
</tr>
<tr>
<td>Conditional ((\beta_1 e_1 \lor \beta_2 e_2))</td>
<td>(\text{&lt;T&gt;} \longrightarrow \text{&lt;START&gt;} \longrightarrow \text{&lt;transition&gt;} \longrightarrow [\varphi(e_2)])</td>
<td>[(e_2)] [(\beta_1)] [(\beta_2)]</td>
</tr>
<tr>
<td>Optional ([e])</td>
<td>(\text{&lt;T&gt;} \longrightarrow \text{&lt;START&gt;} \longrightarrow \text{&lt;transition&gt;} \longrightarrow \text{&lt;START&gt;})</td>
<td>[(e)]</td>
</tr>
<tr>
<td>Iteration ((\beta e)^*)</td>
<td>(\text{&lt;transition&gt;} \quad \text{&lt;transition&gt;})</td>
<td>[(\beta)] [(e)] [(\varphi(e))] [(\beta)] [(e)] [(\text{final})]</td>
</tr>
</tbody>
</table>
Method specifications. In TBSM, a method accomplishes its parent task. Generally, a method consists of a collection of subtasks, a guard condition to invoke the method, and a TSE to specify the temporal relationship among those subtasks. Consider the method in Figure 1b. The TSE is local; no follow operator is allowed, which implies that all the inputs for any demon must be marked. Both $\beta_1$ and $\beta_3$ are treated as propositions. The formula $\beta_1 \lor \beta_3$ is expressed as its equivalent $\neg(\neg\beta_1 \land \neg\beta_3)$. Figure 11 describes the conceptual graph for the method.

TSE composition. To map the composition operator, we use the canonical formation rules: copy, join, simplify, and restrict. The original restrict operator restricts a general concept to a more specific one—for example, restricting the concept [People] to the concept [Man]. We also extend the restrict operator to the graph of expressions, so that we can restrict a graph of expressions with a follow operator to a more specific graph expression based on the idea of specificity rooted in the TSE operators (for example, restricting EXPRESSION(e) to EXPRESSION(c), under the assumption that EXPRESSION(c) conforms to EXPRESSION(e)). To illustrate the transformation, we’ll now examine two of the general cases we described earlier.

Case 1. $T \not\not \in T$ if $T$ does not contain $T$

Because the two graphs to be composed do not have any task in common (the [START] concepts in both graphs are not the same, because they are unknown states), it is trivial to show that the original graphs are equivalent to the disjunction of the two TSE conceptual graphs (see Figure 12).

Case 2. $(\alpha, T, \beta_1; \gamma, \delta; \omega) = (\alpha, T, \beta_1; \gamma, \delta)$

To compose these two graphs, we first locate concepts that are common to both graphs (see Figure 13a). There are five common concepts: $[T]$, $[e(T)]$, $[\beta_1]$, $[\gamma_1]$, and $[\beta_2]$. So, we apply the join operator to these concepts (see Figure 13b). Second, we apply the simplify operator to clean the duplicate $<$transition$>$ demons (see Figure 13c). Third, there are two possible transitions after the concept [e(T)]. One of the transitions is blocked because of an unmarked
input token \( o \), while the other is already enabled \( \gamma_0 \). We apply the restrict operator to restrict the expression with the blocked transition (that is, \( o \)) to a more specific one. In other words, a demon with the input token (that is, \( \gamma_0 \)) and the state after the transition (that is, \( \psi(\gamma_0) \)) are inserted before the blocked transition. The blocked transition indicates a possible sequence of unspecified tasks before the unmarked task \( o \). Similarly, for the expression graph, before enabling the input token \( T! \), we apply the restrict operator to restrict the \( \text{START} \) state (an unknown state) to the state of \( \psi(\alpha) \) (see Figure 13d).

After we apply the restrict operator, there are three common concepts: \( \gamma_1 \), \( \psi(\gamma_0) \), and \( \psi(\alpha) \). Now, we apply the join operator again (see Figure 13e). Finally, we perform the simplify operator to delete duplicate \( \text{final} \) relations and \( \text{transition} \) demons (see Figure 13f).
sent the semantics of constraints satisfaction (that is, the relationships among state objects).

Second, requirements specifications for different views are represented in their conceptual graph specifications, and are tightly, uniformly integrated under the notion of a task and its structure. Also, artifacts constructed in each model (that is, domain, state, functional, or behavioral models) are sharable. For example, constraints in domain models can describe state objects, which in turn often describe functional specifications.

For our future research, we plan to extend the current framework to include fuzzy logic for imprecise requirements, and use conceptual graphs as a basis for requirements acquisition.

**Acknowledgment**

We thank John Yen for his fruitful discussions on an early draft of this article. An earlier version of this article appeared as “Task-Based Specifications through Conceptual Graphs,” Proc. 11th Conf. Artificial Intelligence for Applications, IEEE Computer Society Press, Los Alamitos, Calif., pp. 196–202.

**References**


Related work

Researchers have adopted conceptual graphs as a general knowledge representation formalism for a variety of tasks.

Harry S. Delugach used conceptual graphs to support a multiple-viewed framework that contains an entity relationship diagram, a data flow diagram, a state transition diagram, and a software requirements engineering methodology. Requirements of large software systems might originate with many people. Each person might use a different notation to express his or her view of the requirements. Conceptual graphs can be an internal knowledge representation that can capture the information expressed in several notations. Compilation and extraction algorithms translate the existing requirements languages to and from the internal form, respectively. Developers can then view requirements externally, using different requirements languages for different purposes. Conceptual graphs can also be a formal basis to automatically analyze the resulting multiple viewed requirements.

The Formal Object-Oriented Requirements Model (FORM) used conceptual graphs to provide formal definitions of object-oriented model primitives. A conceptual model represents the problem domain in such a way as to provide a modeling formalism. The symbolic-logic basis of conceptual graphs helps validate and verify FORM both syntactically and semantically. Also, because knowledge is represented in a domain knowledge base and in FORM can be translated into the same formalism, formal deduction validation and verification of FORM with domain knowledge is possible.

Percy L. Loceopoulos and Ralph P. M. Champion used informal models to help elicit and analyze concepts about the application domain. They also argued that conceptual graphs provide the necessary foundation for capturing and analyzing concepts about an application domain. Similarly, Kevin Ryan and Brian Mathews proposed that conceptual graphs can aid the acquisition of requirements. They developed RocColl, a tool that mimics the pattern-filling approach of analyzing the use of modeled patterns for application domains. RocColl uses conceptual graphs to represent concepts acquired and to recognize new concepts through the use of a matching facility.

James R. Stigle, David A. Gardiner, and Kyungsook Han used conceptual graphs to specify a heuristic classification expert system. Their approach identifies six kinds of conceptual structures: type definitions, rules of reasoning, implications, facts, a type lattice, and canonical graphs. Their methodology is suitable for heuristic classification systems whose knowledge can be specified by an inference network that relates pieces of evidence to hypotheses.

Guy Dug-Cross and Michael E. Pilke proposed a conceptual graph-based system to provide a flexible, powerful knowledge engineering approach to building a knowledge base of general concepts. They argued that five features make conceptual graphs useful for research on large, complex, explicit knowledge bases:

- Foundational repositories of concepts and relations,
- Assimilation via graph operations,
- Abstract generalizations and other linguistic structures,
- Confluence and context connectors to build larger structures, and
- Actors to attach procedures to concepts.

References


