Verifying task-based specifications in conceptual graphs

Jonathan Lee*, Lein F. Lai

Software Engineering Lab., Department of Computer Science and Information Engineering, National Central University, Chungli 32054, Taiwan, ROC

Received 28 August 1995; revised 3 November 1997; accepted 13 November 1997

Abstract

A conceptual model is a model of real world concepts and application domains as perceived by users and developers. It helps developers investigate and represent the semantics of the problem domain, as well as communicate among themselves and with users. In this paper, we propose the use of task-based specifications in conceptual graphs (TBCG) to construct and verify a conceptual model. Task-based specification methodology is used to serve as the mechanism to structure the knowledge captured in the conceptual model; whereas conceptual graphs are adopted as the formalism to express task-based specifications and to provide a reasoning capability for the purpose of verification. Verifying a conceptual model is performed on model specifications of a task through constraints satisfaction and relaxation techniques, and on process specifications of the task based on operators and rules of inference inherited in conceptual graphs.

Keywords: Verification; Conceptual model; Task-based specifications; Conceptual graphs

1. Introduction

It is widely recognized that conceptual modeling is an important step towards the construction of users’ requirements [1,19,21]. A conceptual model is a model of real world concepts and application domains as perceived by users and developers [9,23]. It helps developers investigate and represent the semantics of the problem domain, as well as communicate among themselves and with users. Furthermore, conceptual models specify both the static and the dynamic properties of the problem domain in order to serve as a requirements specification for later stages of software development, that is, these models provide an important basis for the design, prototyping and implementation, and against which the design and implementation can be tested [10,16].

There are several benefits for performing the verification at the level of the conceptual model. First, it is easier for developers and users to inspect and validate problem domain knowledge at the conceptual level, since the meaning of problem domain knowledge is not obscured by issues of implementation. Second, the quality and the reliability of end products depend greatly on the accuracy of their conceptual models, as all products originate from these models [15,20]. Finally, verifying a conceptual model helps detect errors early in the software life cycle, thus helping to eliminate design failures and reduce product cost [2,25].

As a result, several works related to the construction and verification of conceptual models have been proposed.

Borgida argued that, in order to express the information in a conceptual model, there is a need for a language [3]. After analyzing several conceptual modeling languages, he has identified six important features that a conceptual modeling language should have: mapping entities in the world to objects, distinguishing an object from its name, performing relations on objects and not on their identifiers, expressing most relationships as properties of objects, grouping objects into classes, and organizing classes into subclass hierarchies.

Lindland et al. proposed a framework to explore the quality of conceptual models [16]. There are three types of model quality: syntactic quality, semantic quality, and pragmatic quality. The goal for syntactic quality is syntactic correctness, which can be achieved by error prevention, error detection, and error correction. As validity and completeness cannot be achieved totally, the semantic goals are decomposed into several subgoals: correct, minimal, annotated and traceable, consistency, and unambiguity. These semantic goals are achieved by introducing statements, deleting statements, correcting statements, and detecting inconsistency. The goal for pragmatic quality is to achieve comprehension through visualization, explanation, simulation, and filtering.
Kung [9] advocated that a good conceptual modeling approach should (1) be a visual and formal approach which is capable of modeling both the static and dynamic aspects of the problem domain, (2) be able to describe the real world semantics incrementally, (3) have a mathematical basis, and (4) produce an executable specification.

The focus of this paper is on the use of task-based specifications in conceptual graphs (TBCG) [13] to construct and verify conceptual models. To construct a conceptual model, task-based specification methodology [14,27] is used to serve as the mechanism to structure the knowledge captured in conceptual models; whereas, conceptual graphs (CGs) are adopted as the formalism to express task-based specifications and to provide a reasoning capability for the purpose of verification. In addition to the essential features identified by Borgida [3], there are several other features inherited in TBCG that are useful for improving the quality of the conceptual model:

- Conceptual graphs enhance the syntactic quality in which the selectional constraints of conceptual graphs facilitate error prevention, while canonical graphs, type definitions, and the type lattice help to accomplish both error detection and error correction.

- The semantic quality can be enhanced by TBCG as follows: first, the use of CGs together with task-based specifications in specifying software requirements helps in capturing rich semantics of the static and dynamic properties of the problem domain. Second, task state expressions (TSE) of TBCG make easy the traceability of the conceptual model. Third, TBCG provides two ways, including first-order logic and CG operators and rules of inference, to verify conceptual models. First-order logic helps to automate the consistency checking; whereas CG operators and rules of inference prompt a more powerful reasoning capability for the purpose of verification. Finally, the formal foundation of TBCG ensures the unambiguity of the conceptual model.

- TBCG enhances the pragmatic quality by providing a graph-based notation to support visualization, an explanation mechanism to describe task-based specifications in conceptual graphs [12], and task state expressions to achieve the simulation.

The verification of task-based specifications in conceptual graphs is performed on: (1) model specifications of a task by checking the consistency of constraints networks through the constraint satisfaction algorithm, and by repairing constraint violation using the constraint relaxation method; and (2) process specifications of a task by checking the interlevel and intralevel consistency based on operators and rules of inference inherited in conceptual graphs.

In the sequel, we first give an overview of task-based specifications in conceptual graphs, which is illustrated using the problem domain of R1/SOAR [22]. In Section 3, the verification for both model specifications and process specifications of tasks is proposed. Finally, we summarize the potential benefits of the proposed approach and outline our future research plan.

2. Task-based specifications in conceptual graphs

Task-based specification methodology acquires and organizes domain knowledge, functional requirements, and high-level problem solving methods around the general notion of tasks. A specification can be described at various abstraction levels, and thus pieces of abstract specification can be refined into a more detailed one in a lower abstraction level. The specification has two components: a model specification that describes static properties of the system and a process specification that characterizes dynamic properties of the system. The static properties of the system are described by two models: a model about domain objects, and a model about the problem-solving states which we refer to as a state model. The dynamic properties of the system are characterized by (1) using the notion of state transitions to explicitly describe what the functionality of a task is, and (2) specifying the sequence of subtasks and interactions between subtasks (i.e. behavior of a system) using task state expressions (TSE). A TSE uses the following operators that can be divided into three groups: (1) sequencing (follow operator and immediately follow operator), (2) branching (selection, optional operator and conditional operator), and (3) iteration (iteration operator). Both the model and the process specifications can be first described in their high-level abstract forms, which can be further refined into more detailed specifications at the next level. The notion of task structure (i.e. task/method/subtask) [4] is adopted for the process refinement. A more detailed description of TBSM can be found in Refs. [11,14,27].

The conceptual graph [24] is a directed, finite, connected graph and consists of concepts, concept instances (referents) and conceptual relations. Concepts and relations represent declarative knowledge. Procedural knowledge can be attached through actors. Actors represent processes that can change the referents of their output concepts, based on their input concepts. Concepts are represented in square brackets, relations in parentheses, and actors in angle brackets. Delugach has extended conceptual graphs to include a new type of node, demons (in double angle brackets), to cause creation and retraction of input and output concepts [5]. A demon’s algorithm causes each of its actual output concepts with referents to be asserted (i.e. marked), while each of its actual input concepts is to be retracted. If there is more than one input concept, no demon action occurs until all of its input concepts have been asserted. The notion of constraint overlays has also been incorporated into the conceptual graphs, which provides a method to attach constraints to objects and collections of objects that make up world states [6,18]. Constraint
overlays, represented in angle brackets and linked to concepts with dash lines, can overlay actors (procedures or constraints) on a conceptual graph to describe the changes to a model state.

Conceptual graphs have several useful 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 model in our methodology. An overview of the proposed approach depicting the mapping from task-based specifications to conceptual graphs is shown in Fig. 1. Details of task-based specifications in conceptual graphs can be found in Ref. [13].

The following example, which is adopted from R1/SOAR [27], illustrates various components of task-based specifications in conceptual graphs.

R1/SOAR focuses on the unibus configuration portion of R1’s configuration task. Two major tasks of concern in this example are configure modules and configure a module. The functionality of the task configure modules is to configure as many modules as possible into the current backplane while maintaining the optimal ordering in which modules are configured. This is achieved by iterating between two subtasks: (1) configure a module, which attempts to configure the current module into the current backplane, and (2) 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 under the situation that the current module fails to be configured into the current backplane, since we have not finished configuring the current module yet. It is also desirable to indicate that, if the current module cannot be configured into the current backplane, the system should later find a backplane suitable for the current module before configuring the next module. The specifications in conceptual graphs for the tasks and methods relevant to this example are shown in Figs. 2–4.

2.1. Domain model

Since models are similar to entity-relationship models, the transformation of terms, attributes and relations to conceptual graphs is straightforward. First, a term becomes a concept of type ENTITY. Second, an attribute associated with a term is represented by a relation (attr), and an attribute name is characterized by a concept type as a data type through a relation (chrc). Finally, a relation becomes a conceptual relation. Consider the task specification of Configure-Modules in Fig. 2, the domain model expresses two terms (i.e. Module and Backplane) with their attributes (i.e. Power, Volume, and Room) and two constraints (i.e. $T \leq L_1$ and $T \leq L_2$). The referent of each concept attaches an ‘@’ to indicate the cardinality of the concept.

2.2. 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. Our translation rules are summarized as follows: (1) to overlay constraint actors on the conceptual graph of state objects. The constraint overlays are used to show the relationships among state objects. All state objects associated with a constraint actor are required to satisfy the relationship expressed by the actor. (2) To use demons for state transitions whose inputs and outputs are state objects of the type STATE, and 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. For example (see Fig. 2), in the task specification of Configure-Modules, the state model expresses that the state [Backplane]. [Module].] will be changed into another state [Moduled-Backplane] under the conditions that (1) constraints \( T_1 \) and \( T_2 \) are satisfied, and (2) the task Configure-Modules is performed.

2.3. Functional specifications

Preconditions, protections and rigid postconditions are

Fig. 3. A method of the task Configure-Modules.

Fig. 4. The task Configure-a-Module.
represented by propositions. A soft postcondition can be viewed as a fuzzy proposition with fuzzy concepts and fuzzy conceptual relations. Consider the task Configure-Mdoules in Fig. 2, the precondition of the task represented by a proposition says that a backplane and a set of unconfigured modules exists.

2.4. Behavioral specifications

TSE is an extension of regular expressions and, therefore, can be represented using state transition diagrams. To transform TSEs into conceptual graphs, we have adopted the notion of demons to represent transitions, which possesses the semantics of an actor node with respect to output concepts’ referents, with the additional semantics that a demon actually asserts its output concepts and then retracts its input concepts [5]. We also assume that there is a mapping \( \varphi \) that maps an expression to its state where the postcondition after progressing through the expression is true. The distinction between follow and immediately follow operator is noted by using the demon for the immediately follow operator with the task before and after the operator being marked, whereas the follow operator is transformed into a demon whose inputs are not yet completely marked.

In the cases of selectional, iteration, conditional and optional operators, two conventions are adopted. First, we follow the tradition of demons by using an initiator demon (i.e. \([\text{START}]\)) and a \text{START} concept (a subtype of \text{STATE}) for the beginning state. Second, the convention of viewing a conditional test as a task (e.g., see Ref. [26]) is also adopted, denoted as \( \beta_t \), where \( \beta \) is a condition. That is, \( \beta \) is a special control flow task that is invoked only when \( \beta \) is tested to be true, whereas \( \neg \beta \) is another special control flow task that is invoked only when \( \neg \beta \) is tested to be true. A final state is denoted by attaching the monadic relation \( \text{final} \). The difference between selectional and conditional operators is that the expression \( \epsilon \), \( \beta \) will not be performed unless the conditional test for \( \beta \) is tested to be true, neither of which are marked. The optional operator is treated as a special case of selectional operator. The iteration operator is implemented using three demons. The first demon is invoked by performing an expression \( \epsilon \), while the second demon will be invoked by two input tokens, \( \beta \) and \( \epsilon \), to represent the notion of iteration condition. The third demon is to indicate the exit condition. The mapping of TSE operators to conceptual paths is summarized in Table 1.

### Table 1

Mapping TSE operators to conceptual paths

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>In Conceptual Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediately Follow</td>
<td>([T_1, T_2])</td>
<td>([T_1, T_2]), ([T_1, T_2])</td>
</tr>
<tr>
<td>Follow</td>
<td>([T_1, T_2])</td>
<td>([T_1, T_2]), ([T_1, T_2])</td>
</tr>
<tr>
<td>Selectional</td>
<td>({[\text{STATE}],[\text{STATE}]} )</td>
<td>({[\text{STATE}],[\text{STATE}]} )</td>
</tr>
<tr>
<td>Conditional</td>
<td>({[\text{STATE}],[\text{STATE}]} )</td>
<td>({[\text{STATE}],[\text{STATE}]} )</td>
</tr>
<tr>
<td>Optional</td>
<td>({[\text{STATE}]} )</td>
<td>({[\text{STATE}]} )</td>
</tr>
<tr>
<td>Iteration</td>
<td>({[\text{STATE}]} )</td>
<td>({[\text{STATE}]} )</td>
</tr>
</tbody>
</table>

2.5. Method specifications

One of the methods in TBSM is to accomplish 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.

3. Verification

Verifying a conceptual model helps detect errors early in the software life cycle, thus helping to eliminate design failures and reduce product cost. The verification of task-based specifications in conceptual graphs is performed on: (1) model specifications of a task by checking the consistency of constraint networks [8,17] through the constraint satisfaction algorithm [7], and by repairing constraint violation using the constraint relaxation method; and (2) process specifications of a task by checking the interlevel and intra-level consistency based on operators and rules of inference inherited in conceptual graphs [24]. The following algorithm will verify task specifications in its conceptual graphs one task at a time from the highest level task:

### Algorithm 1
To verify a task specification \( T \) in its conceptual graphs:

1. Verifying the model specification \( T \) by:
(a) Establishing a constraint hierarchy for all constraints in \( T \).
(b) Building the constraint network level-by-level based on the strength of constraints.
(c) Checking the consistency of constraint networks by the constraint satisfaction algorithm and repairing constraint violation by the constraint relaxation method.

2. Verifying the process specification of \( T \) by:
   (a) Comparing the process specification of a task \( T \) with the method that contains \( T \) (called \( T \)'s parent method) and other subtasks in the method to prove or disprove \( T \)'s precondition using Beta rules of inference.
   (b) Comparing the process specification of a task \( T \) with those methods that accomplish \( T \) (called \( T \)'s child method) and their subtasks for consistency checking.
   (i) The before state description of \( T \)'s child method must be semantically more specific than \( T \)'s precondition.
   (ii) The protection of \( T \) is not violated by any subtasks of \( T \).
   (iii) The after state description of \( T \)'s child method must be semantically more specific than \( T \)'s rigid post-condition.

3.1. Verifying Model Specifications

A constraint hierarchy can be established based on the strength (denoted as \( C_0, \ldots, C_3 \)) associated with each constraint. The constraint hierarchy is useful for reasoning about constraints using constraint satisfaction and relaxation techniques [7]. A constraint network can be built level-by-level from constraints at the top level of the hierarchy down to the lower one. The constraint network [8,17], which is useful for checking the consistency among constraints, can be established for all the constraints in a task by: (1) examining the dependency relationship between their inputs and outputs; and (2) viewing the relationships among concepts as predicates with TRUE value.

Constraints with the greatest (denoted as \( C_0 \)) strength are first added to the conceptual graph. These \( C_0 \) constraints cannot be violated and an initial constraint network can thus be established. If any violation occurs, their fix rules will be used to repair the situations. If all \( C_0 \) constraints are consistent, the \( C_{i+1} \) (weaker than \( C_i \)) constraints would be added to the constraint network and then the consistency would be checked in turn. Weak constraints can be violated. If any inconsistency occurs, relaxation will be applied and constraints with weakest strength will be removed. After all constraints are applied, the constraint network would be complete.

Verifying the task specification in its conceptual graphs can be formulated as a constraint satisfaction problem by first treating concepts as a set of variables, each of which must be instantiated in a particular domain and by considering constraints as predicates and relations as predicates with only TRUE value.

Consider the domain model of the task Configure-a-Module, two \( C_3 \) constraints are: (1) the module’s pin type is equal to the backplane’s pin type; and (2) the module’s pin type is KMC11. One of the \( C_1 \) constraints is that the backplane’s pin type is RK611. The conceptual graph of the domain model of Configure-a-Module and its logical form are described below.

\[
[\text{Module}] \rightarrow \text{attr} \rightarrow [\text{Pin}] \rightarrow \text{attr} \rightarrow [\text{Pin} \rightarrow \text{Type}]
\]

\[
[\text{Backplane}] \rightarrow \text{attr} \rightarrow [\text{Pin}] \rightarrow \text{attr} \rightarrow [\text{Pin} \rightarrow \text{Type}]
\]

\[
(\exists x_1)(\exists x_2)(\exists y_1)(\exists y_2)(\exists y_3) \text{Module}(x_1) \\
\land \text{attr}(x_1,x_2) \land \text{Pin}(x_2) \land \text{attr}(x_2,x_3) \\
\land \text{Pin} \rightarrow \text{Type}(x_1) \land \text{Backplane}(y_1) \land \text{attr}(y_1,y_2) \\
\land \text{Pin}(y_2) \land \text{attr}(y_2,y_3) \land \text{Pin} \rightarrow \text{Type}(y_1).
\]

The constraint hierarchy can thus be established as follows. (1) \( C_0 \): equal-to(module’s pin type, backplane’s pin type), equal-to-KMC11(module’s pin type). (2) \( C_1 \): equal-to-RK611(backplane’s pin type).

To verify if all the constraints are consistent, we first consider those constraints with \( C_3 \) strength to be added to the conceptual graphs. Dash lines denote the constraints:

\[
\text{equal-to-KMC11}
\]

\[
[\text{Module}] \rightarrow \text{attr} \rightarrow [\text{Pin}] \rightarrow \text{attr} \rightarrow [\text{Pin} \rightarrow \text{Type}].
\]

\[
[\text{Backplane}] \rightarrow \text{attr} \rightarrow [\text{Pin}] \rightarrow \text{attr} \rightarrow [\text{Pin} \rightarrow \text{Type}].
\]

The initial constraint network consists of two predicates of constraints:

\[
(\exists x_1)(\exists y_3) \text{equal} \rightarrow \text{to}(x_1,y_3) \text{ and equal} \rightarrow \text{to} \rightarrow \text{KMC11}(x_1).
\]

In examining the node consistency, we check the unary predicate equal-to-KMC11(\( x_1 \)). The domain of \( x_1 \) would be narrowed down and contains only KMC11. The domain is not a null set, which indicates that the node consistency is not violated. In verifying the arc consistency, we check the binary constraint equal-to(\( x_1, y_3 \)). The domain of \( y_3 \) would be restricted to contains only KMC11, thus the arc consistency is also ensured. All relations among concepts are viewed as predicates with TRUE value, therefore, we know that the path consistency is not violated. After applying the constraint propagation algorithms (including node, arc and path consistency) on the two predicates, we found that no inconsistency was occurred. That is, no relaxation is required. Subsequently, the constraint network is updated by adding the constraint equal-to-RK611 with the strength of \( C_1 \).
After trying domains of Definition 1, rules of inference are described below.

To examine the node consistency, we check the unary predicate equal-to-KMC11 and equal-to-RK611. The domain of \( x \) would contain only KMC11 and the domain of \( y \) would contain only RK611. In verifying the arc consistency, we check the binary constraint equal-to($(x, y)$). After trying domains of \( x \) and \( y \), none of the instantiations can satisfy the binary constraint. As a result, we have detected an arc inconsistency in the constraint network. Therefore, we relax the constraint network by removing the weakest constraint (the \( C_1 \) constraint). The final constraint network would be consistent after checking all levels of constraints.

### 3.2. Verifying process specifications of a task with its parent method

Process specifications of a task, \( T \), is consistent with its parent method if the precondition of the task can be deduced from the state description before performing the task. That is, \( T \)'s precondition must be provable from \( T \)'s before state description. Sowa’s Beta rules of inference [24], a generalization of Peirce’s Alpha rules, provide a proper reasoning capability to prove one conceptual graph from another. The Beta rules of inference are described below.

**Definition 1.** Let the outermost context contain a set \( S \) of conceptual graphs. Any graph derived from \( S \) by the following first-order rules of inference is said to be provable from \( S \).

- **Erasure.** In an evenly enclosed context, any graph may be erased, any coreference link from a dominating concept to an evenly enclosed concept may be erased, any referent may be erased, and any type label may be replaced with a supertype.
- **Insertion.** In an oddly enclosed context, any graph may be inserted, a coreference link may be drawn between any two identical concepts, and restriction may be performed on any concept.
- **Iteration.** A copy of any graph \( u \) may be inserted into the same context in which \( u \) occurs or into any context dominated by \( u \). A coreference link may be drawn from any concept of \( u \) to the corresponding concept in the copy of \( u \). If concepts \( a \) and \( b \) in some context \( c \) are both dominated by a concept \( d \) on some line of identity, then a coreference link may be drawn from \( a \) to \( b \).
- **Deiteration.** Any graph or coreference link whose occurrence could be the result of iteration may be erased. Duplicate conceptual relations may be erased from any graph.
- **Double negation.** A double negation may be drawn around or removed from any graph in any context.
- **Coreference join.** Two identical, coreferent concepts in the same context may be joined, and the coreference link between them may then be erased.
- **Individuals.** If an individual concept \( a \) dominates a generic concept \( b \) where \( a \) and \( b \) are coreferent, then referent(a) may be copied to \( b \), and the coreference link may be erased.

Consider the example in R1/SAOR, \( T_1 \)'s before state description (i.e. BSD1) is the conjunction of the guard condition and the before state of its parent method (see Fig. 5). \( T_2 \)'s before state description (i.e. BSD2) is the result of BSD1 progressing through the TSE of the parent method before \( T_2 \) is invoked. To verify the consistency, we need to check if the preconditions of \( T_1 \) and \( T_2 \) are provable from their BSDs. The precondition of \( T_2 \) is described below.

\[
G_1 : \{\text{Backplane} \}, \{\text{Module} \} \rightarrow \{ \ast \} \\
\rightarrow (\text{attr}) \rightarrow [\text{Unconfigured}]
\]

The guard condition of the parent method is ‘True’ (i.e. the universal concept \( [T] \) in conceptual graphs), and the TSE of the parent method is:

\[
(T_1, (\beta_1 \land \beta_2)T_2 \lor \neg(\beta_1 \land \beta_2))^{*}
\]

From the TSE, we know that the first task invoked is \( T_2 \); therefore, the before state of the parent method is equal to \( T_2 \)'s precondition (i.e. \( G_2 \)). The guard condition (i.e. \( [T] \)) and the before state of the parent method (i.e. \( G_2 \)) are combined into BSD1 (i.e. \( G_2 \)).

\[
G_2 : \{\text{Backplane} \}, \{\text{Module} \} \rightarrow \{ \ast \} \\
\rightarrow (\text{attr}) \rightarrow [\text{Unconfigured}],[T]
\]

To examine whether \( G_1 \) is provable from \( G_2 \), we use Beta

![Fig. 5. The task Configure-a-Module with its parent method.](image-url)
rules of inference to check if \( G_1 \) can be derived from \( G_2 \). By applying the *Erasure* rule to erase \( \{T_i\} \), we can deduce \( G_1 \) from \( G_2 \). Since \( T_j \)'s precondition (i.e. \( G_3 \)) is provable from its before state description (i.e. \( G_2 \)), the task *Configure-a-Module* is consistent with its parent method.

Next, consider the task *Obtain-the-Next-Module* (\( T_j \)). In order to obtain \( BSD_2 \), we first compute \( BSD_1 \) and then progress \( BSD_2 \) through the TSE of the parent method. Based upon the TSE, \( T_j \) must be performed before \( T_j \) is invoked. Therefore, \( BSD_2 \) can be computed by progressing \( BSD_1 \) through the task *Configure-a-Module*. The postcondition of \( T_j \) is:

\[
G_2 \vdash [\text{Module}] \implies [\text{Backplane}] \implies\{\text{state}\} \implies [\text{Module-Backplane}] \implies [\text{attr}] \implies [\text{Unconfigured}] \}
\]

To progress \( BSD_2 \) through the task *Configure-a-Module*, \( BSD_2 \) (i.e. \( G_2 \)) and \( T_j \)'s postcondition (i.e. \( G_3 \)) are joined. Applying Sowa’s *join* operator on two concepts [Backplane] and [Module], \( G_4 \) is derived.

\[
G_4 \vdash [\text{Module}] \implies [\text{compatible}] \implies [\text{configured-into}] \implies [\text{Backplane}] \implies [\text{state}] \implies [\text{Module-Backplane}] \implies [\text{attr}] \implies [\text{Unconfigured}] \}
\]

As the condition \( \beta_1 \land \beta_2 \) should be satisfied before \( T_j \) is invoked, \( BSD_2 \) (i.e. \( G_4 \)) is obtained by combining \( G_4 \) and \( \beta_1 \land \beta_2 \) (see Fig. 6). The conceptual graph of \( \beta_1 \land \beta_2 \) is described below.

\[
G_4 \vdash [\text{Module}] \implies [\text{compatible}] \implies [\text{configured-into}] \implies [\text{Backplane}] \implies [\text{state}] \implies [\text{Module-Backplane}] \implies [\text{attr}] \implies [\text{Unconfigured}] \}
\]

The precondition of \( T_j \) is:

\[
G_5 : \{\text{Backplane}\} \rightarrow \{\text{state}\} \rightarrow \{\text{Module-Backplane}\}.
\]

To examine whether \( T_j \) is consistent with its parent method, it is required to check if \( T_j \)'s precondition (i.e. \( G_5 \)) is provable from \( BSD_2 \) (i.e. \( G_6 \)). By applying the *Erasure* rule to erase \( \{\text{Module}\} \), \( \{\text{compatible}\} \), and \( \{\text{configured-into}\} \), we can derive \( G_2 \) from \( G_6 \). Therefore, the task *Obtain-the-Next-Module* is consistent with its parent method.

### 3.3. Verifying process specifications of a task with its child method

There are three steps involved in verifying if a child method is consistent with its task \( T \): (1) the before state description of a \( T \)'s child method must be semantically more specific than \( T \)'s preconditions; (2) the protection of \( T \) is not violated by any subtasks of \( T \); and (3) the after state description of a \( T \)'s child method must be semantically more specific than \( T \)'s rigid postcondition. Based on Sowa’s definition [24], we define the notion of specificity below.

**Definition 2.** A conceptual graph \( u \) is more specific than \( v \) if and only if \( u \) is canonically derivable from \( v \), where \( u \) is called a specialization of \( v \), denoted as \( u \leq v \), and \( v \) is called a generalization of \( u \).

Several important implications are derived from this definition: any subgraph is a generalization of the original; replacing a type label with a supertype generalizes a graph; and erasing an individual marker generalizes a graph. In particular, the graph consisting of the single universal concept \( \{T\} \) is a generalization of every other conceptual graph.

Consider the task *Configure-Modules* (\( T \)) with its child method and two subtasks \( T_j \) and \( T_k \) (see Fig. 7). \( BSD \) and \( ASD \) denote the before and after state descriptions of its child method, respectively. Based upon the TSE of its child method, \( BSD \) is the conjunction of the guard condition

\[
G_6 : \{\text{Backplane}\} \rightarrow \{\text{Module}\} \rightarrow \{\text{attr}\} \rightarrow \{\text{Unconfigured}\} \}
\]

and the precondition of the first task. \( ASD \) is equal to the result that \( BSD \) progresses through the child method’s TSE.

We first check if \( BSD \) is more specific than the precondition of \( T \). Since the first task invoked by the TSE is \( T_j \), \( BSD \) is equal to the conjunction of \( T_j \)'s precondition and the guard condition (i.e. \( G_5 \)).

\[
G_5 : \{\text{Backplane}\} \rightarrow \{\text{Module}\} \rightarrow \{\text{attr}\} \rightarrow \{\text{Unconfigured}\} \}
\]

The conceptual graph of \( T \)'s precondition is:

\[
G_8 : \{\text{Backplane}\} \rightarrow \{\text{attr}\} \rightarrow \{\text{Unconfigured}\} \}
\]
we can project $G_9$ onto $G_8$. Since $\{\text{Backplane}\}$ is a projection of $G_9$ in $G_8$, we can obtain $G_8 \preceq G_9$ (i.e. BSD is more specific than $T$'s precondition). Hence, we know that the precondition of the task $\text{Configure-Modules}$ is consistent with the before state description of its child method.

To ensure that $T$'s protection is not violated by any subtasks of $T$, all conditions (i.e. precondition, protection, and postcondition) of $T_1$ and $T_2$ must be consistent with $T$'s protection. The consistency checking mechanism of conceptual graphs can be exploited to examine whether two conceptual graphs are consistent. The definition of consistent is described below.

**Definition 3.** A set $S$ of conceptual graphs is said to be consistent if there is no pair of conceptual graphs $p$ and $\neg[p]$ that are both provable from $S$.

$T$'s protection and $T_1$'s precondition are $G_{10}$ and $G_{11}$, respectively.

$G_{10} : [\text{Module}] \rightarrow \{\text{keep}\} \rightarrow [\text{Optimal} \rightarrow \text{Order}]$.

$G_{11} : [\text{Backplane}].[\text{Module}].[\text{Module} : \{+\}] \rightarrow \{\text{attr}\} \rightarrow [\text{Unconfigured}]$.

Since $G_{10}$ and $G_{11}$ cannot derive any pair of conceptual graphs $p$ and $\neg[p]$, $G_{10}$ and $G_{11}$ are consistent. After examining all conditions of $T_1$ and $T_2$, we can ensure that all the conditions are consistent with $G_{10}$. Therefore, $T$'s protection is not violated by any subtasks of $T$.

Finally, we check if the ASD of $T$'s child method is more specific than $T$'s postcondition. To obtain the ASD, we first compute the BSD and then progress the BSD through the TSE of its child method. From the TSE, we know that $T_1$ must be performed first. To progress the BSD through the task $\text{Configure-a-Module}$ ($T_1$), BSD (i.e. $G_8$) and $T_1$'s postcondition are joined. Applying the join operator on two concepts $[\text{Backplane}]$ and $[\text{Module}]$, $G_{12}$ is obtained.

![Fig. 6. To derive BSD_2: (a) combining $G_i$ and $\beta_1 \land \beta_2$, (b) applying the Iteration rule, (c) applying the Deiteration rule, and (d) applying the Double Negation operator.](image)

![Fig. 7. The task Configure-Modules with its child method.](image)
Subsequently, it depends on the condition $\beta_1 \land \beta_2$ to decide whether $T_1$ or $T_{exit}$ will be performed. The post-condition of the task \textit{Obtain-the-Next-Module} ($T_2$) is:

\begin{align*}
G_{13} : [\text{Backplane}] &\rightarrow (\text{state}) \rightarrow [\text{Moduled} - \text{Backplane}] ,
\end{align*}

[Module : \{ * \}] $\rightarrow$ (attr) $\rightarrow$ [Unconfigured].

Progressing $G_{12}$ through $T_2$ or $T_{exit}$, ASD is obtained:

\begin{align*}
G_{15} & : [\text{Backplane}] \rightarrow (\text{state}) \rightarrow [\text{Moduled} - \text{Backplane}] ,
\end{align*}

[Module : \{ * \}] $\rightarrow$ (attr) $\rightarrow$ [Unconfigured].

The postcondition of the task \textit{Configure-Modules} ($T$) is:

\begin{align*}
G_{15} : [\text{Backplane}] &\rightarrow (\text{state}) \rightarrow [\text{Moduled} - \text{Backplane}] ,
\end{align*}

[Module : \{ * \}] $\rightarrow$ (attr) $\rightarrow$ [Unconfigured].

To examine the specificity between $G_{14}$ and $G_{15}$, we can project $G_{14}$ onto $G_{15}$. Since $[\text{Backplane}] \rightarrow (\text{state}) \rightarrow [\text{Moduled-Backplane}]$ is a projection of $G_{15}$ in $G_{14}$, we have $G_{14} \cong G_{15}$ (i.e. ASD is more specific than $T$’s post-condition). Hence, we know that the postcondition of the task \textit{Configure-Modules} is consistent with the after state description of its child method.

In this section, we have applied the proposed verification algorithm to a part of the R1/SOAR specifications. More specifically, we have demonstrated: (1) the verification on the model specification of the task \textit{Configure-a-Module} through constraints satisfaction algorithm; (2) the verification on process specifications of two tasks \textit{Configure-a-Module} and \textit{Obtain-the-Next-Module} with their parent method based on Beta rules of inference; and (3) the verification on the process specification of the task \textit{Configure-Modules} with its child method based on the projection operator inherited in conceptual graphs.

4. Conclusion

In this paper, we propose the use of task-based specifications in conceptual graphs to construct and verify a conceptual model. Task-based specification methodology is used to serve as the mechanism to structure the knowledge captured in the conceptual model; whereas, conceptual graphs are adopted as the formalism to express task-based specifications. The verification of task-based specifications in conceptual graphs is performed on: (1) model specifications of a task by checking the consistency of constraints networks through the constraint satisfaction algorithm, and by repairing constraint violation using the constraint relaxation method; (2) process specifications of a task by checking the interlevel and intralevel consistency based on the projection operator and Beta rules of inference.

conceptual graphs facilitates the capturing of semantics that task-based specifications find difficult to express. For example, the semantics of state transitions (i.e. the notion of retraction and assertion of state objects) can be manifested through demons; whereas the semantics of constraints satisfaction (i.e. the relationships among state objects) can be represented using actor overlays.

- Requirements specifications for different views are represented in their conceptual graphical specifications, and are \textit{tightly integrated} under the general notion of tasks. In addition, artifacts constructed in each model (i.e. domain, state, functional, or behavioral) are sharable, for example, constraints in domain models can be used to describe state objects, which in turn are usually used in the description of functional specifications.

- TBCG uses first-order logic, CG operators, and rules of inference to verify conceptual models. First-order logic helps to automate the consistency checking; whereas, CG operators and rules of inference prompt a more powerful reasoning capability for the purpose of verification. For example, CG rules of inference can derive a generic graph from a restricted one, which is absent from first-order logic.

Our future research plan will consider the following tasks: (1) to extend the current framework to fuzzy logic for modeling imprecise requirements, and (2) to utilize task-based specifications in conceptual graphs as design patterns for reusing requirements specifications.

Acknowledgements

This research was sponsored by National Science Council (Taiwan, ROC) under grant NSC86-2213-E-008-006.

References


