University Timetabling through Conceptual Modeling

Jonathan Lee,1,* Shang-Pin Ma,1,† Lien Fu Lai,2,‡ Nien Lin Hsueh,3,§ Yong-Yi Fanjiang4,¶
1Department of Computer Science and Information Engineering, National Central University, Chungli 320, Taiwan
2Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua 500, Taiwan
3Department of Information Engineering and Computer Science, Feng Chia University, Taichung 400, Taiwan
4Department of Computer and Information Science, National Taichung University, Taichung 403, Taiwan

A number of approaches have been proposed in tackling the timetabling problem, such as operational research, human-machine interaction, constraint programming, expert systems, and neural networks. However, there are still several key challenges to be addressed: easily reformulated to support changes, a generalized framework to handle various timetabling problems, and ability to incorporate knowledge in the timetabling system. In this article, we propose an automatic software engineering approach, called task-based conceptual graphs, to addressing the challenges in the timetabling problem. Task-based conceptual graphs provide the automation of software development processes including specification, verification, and automatic programming. Maintenance can be directly performed on the specifications rather than on the source code; moreover, hard and soft constraints can be easily inserted or removed. A university timetabling system in the Department of Computer Science and Information Engineering at National Central University is used as an illustrative example for the proposed approach. © 2005 Wiley Periodicals, Inc.

1. INTRODUCTION

Timetabling is a major administrative activity for a wide variety of institutions. Timetabling can be defined as a process of assigning events or activities to resources such as time (timeslots), space (rooms), and personnel (teachers or crews),

*Author to whom all correspondence should be addressed: e-mail: yjlee@selab.csie.ncu.edu.tw.
†e-mail: albert@selab.csie.nce.edu.tw.
‡e-mail: lfaii@cc.ncue.edu.tw.
§e-mail: nlhsueh@fcu.edu.tw.
¶e-mail: yyfanj@selab.csie.ncu.edu.tw.

which satisfy all constraints. Over the last 30 years, many approaches and models have been proposed for dealing with a variety of timetabling programs. Most of these works employed operational research, constraint programming, expert systems, or neural networks to solve the timetabling problem. However, there are still several key challenges to be addressed:

- The timetabling problem varies significantly from institution to institution in terms of specific requirements and constraints. The timetabling systems should be easily reformulated or customized to support changes.
- The lack of a generalized framework makes the development of a timetabling system slowly and expensively. A generalized methodology that can handle a wide range of timetabling problems is required.
- It is hard to capture knowledge and incorporate it into the timetabling system. Expertise helps in reducing the search space and in fitting the solution to the context.
- The difference between hard and soft constraints should be addressed. A constraints relaxation technique is needed for obtaining a feasible solution.

In this article, we propose an automatic software engineering approach, called task-based conceptual graphs (TBCG), to addressing these challenges. TBCGs provide the automation of software development processes including specification, verification, and automatic programming. First, we use TBCGs to capture and represent a conceptual model for the problem domain. Second, contextual retrieval mechanism proposes a multiple-viewed approach for helping acquire specifications and elicit information in contexts. Third, the verification of TBCGs is performed on model specifications and process specifications of each task. Finally, we operationalize TBCGs for the execution of task-based specifications. Once TBCGs have been constructed and verified, a blackboard system will automatically transform TBCG specifications into an executable university timetabling program. Major constituents of a software system such as database schemas, the knowledge base, and user interfaces can also be generated by transforming TBCGs into their corresponding counterparts.

We have applied the TBCG approach in timetable planning for the Department of Computer Science and Information Engineering at National Central University in Taiwan. We have also adopted some observations made in Ref. 16 about the timetabling problem to further confine our scope on the problem: First, it assumes that students are already assigned or will select classes for themselves. Second, constraints can be further classified as hard or soft. Finally, the aim of timetabling algorithms is to find a feasible solution that satisfies all hard constraints, and as many soft constraints as possible, rather than to optimize on some chosen objective functions.

The expressive power of TBCGs makes it easier to model the timetabling problem. As a first-order logic vehicle, TBCG is powerful in describing domain knowledge, static facts, constraints, and relationships in the real world. Moreover, TBCG proposes a constraint satisfaction and relaxation mechanism to spot a feasible solution that satisfies all hard constraints and as many soft constraints as possible. With its graphic and formal representation, TBCGs can serve as a readable
and executable specification language. Small logical changes may have large and complex effects while performing maintenance at the implementation level; whereas maintenance at the specification level is fairly simple and explainable. As TBCG specifications can be transformed into executable programs automatically, maintenance can then be directly performed on the specifications rather than on the source code. TBCG hence offers a certain degree of flexibility to support changes.

In the Section 2, we discuss related work. Section 3 gives a problem statement for the timetabling problem we are dealing with. In Section 4, we give an overview on how to express task-based specifications in conceptual graphs, and how to model the university timetabling by using TBCG. We also discuss issues identified and the proposed approach to operationalize TBCGs and the transformation between TBCG and CLIPS. Section 5 outlines how the system is built. Finally, we summarize the proposed approach in Section 6.

2. RELATED WORK

A wide variety of approaches to the timetabling problem have been proposed. These works can be classified into three categories based on the underlying technologies adopted: operational research (OR), human-machine interaction, and artificial intelligence (AI).

2.1. Operations Research

Over the last 30 years, many OR techniques ranging from the use of mathematical programming to heuristics are adopted to solve the timetabling problem. These studies can be roughly divided into three types: sequential methods, cluster methods, and meta-heuristic methods.

Sequential methods order activities based on how difficult it is to schedule them and then assign the activities into valid resources in sequence. Graph coloring is the most common sequential methods to solve the timetabling problem. de Werra shows how to reduce a course timetabling problem into the graph coloring problem. By applying graph coloring technology, the problem can be formulated as a network of arcs and nodes, and the solution can be obtained by finding a set of colors in which no two adjacent nodes or edges have the same color. While these techniques adapt well to small-scale problems, they fail to scale up for large ones and cannot be easily reformulated or customized to support change.

Cluster methods split activities into groups based on whether it satisfies hard constraints and then assign the groups into resources to fulfill the soft constraints. Tripathy employs a Lagrangian Relaxation technique to solve the timetabling problem. In his approach, the number of variables is reduced by grouping, the number of constraints is reduced by replacing some of relaxed constraints, and feasible solutions are obtained by heuristics. Ferland and Roy formulate the timetabling problem as an assignment problem and solves it through a reduction to a quadratic assignment problem. The main drawback of these approaches is that the
clusters of events are formed and fixed at the beginning of the algorithm and that may result in a poor quality timetable.\textsuperscript{30}

Meta-heuristic methods begin with some initial solutions and use search strategies that try to avoid local optima. The memetic algorithm\textsuperscript{5} improves the performance of a genetic algorithm by incorporating a local neighborhood search, which explores the neighborhood of the solutions obtained by a genetic algorithm and navigates the search toward the local optima for each solution. Hertz\textsuperscript{3,31} uses the tabu search technique to solve the university course timetabling problems. Reference 31 considers the optimization problem with the grouping option, whereas Ref. 3 extends the approach to a more complex case that takes into account lectures of different length. Schaerf\textsuperscript{6} proposed an algorithm based on a local search, which is constituted by tabu search and randomized hill-climbing. These methods can produce high-quality solutions but often have a considerable computational cost.\textsuperscript{30}

2.2. Human-Machine Interaction

The human-machine interaction approach is an iterative process. The machine first proposes an initial solution. Subsequently, the user either accepts it, improves upon it manually, or instructs the machine to provide another solution. The process iterates until the user is satisfied, or no further process is possible.

Mulvey\textsuperscript{9} proposes a model of approximation, evaluation, and modification to deal with the timetabling problem. First, the user requires an approximate model of the ill-defined problem. Second, this model has to be evaluated by an estimator as to whether it provides a good starting solution, and how flexible it is in aiding the user to improve upon the solution. Finally, the user or machine goes through the modification process.

Johnson\textsuperscript{8} uses a database management system for the bookkeeping aspects of timetable development. The system alleviates clerical burden associated with the production of timetables. In addition, the provision of accurate information aids the decision making to management on teaching commitments for resource allocation purposes. The major drawback of such systems is that for large problems, the process can be laborious, and obtaining a good initial solution may be computationally expensive.\textsuperscript{16}

2.3. Artificial Intelligence

In recent years, a number of AI technologies, including constraint programming, expert systems, and neural networks, are used for solving the timetabling problems.

Constraint programming approaches formulate the timetabling problem as a constraint satisfaction model and then propagate constraints to obtain a feasible solution. Monfroglio\textsuperscript{13} proposes a Prolog-based system that employs backtracking for finding feasible timetables. The system decomposes and classifies constraints with respect to message passing and the constraint ordering to minimize the backtracking and maximize the parallelism. Deris et al.\textsuperscript{11} propose a
constraint-based reasoning algorithm to model and solve the timetabling problem. The proposed system is implemented in an object-oriented approach and can thus be easily adapted to support changes. In Ref. 14, OR models and local search techniques are also used to assist the constraint programming search process by effectively reducing the solution search space. They propose a minimum cost matching algorithm to relax the constraint satisfaction model. Constraint logic programming\textsuperscript{10,12} integrates logic programming and constraint solving to tackle combinatorial problems such as planning, scheduling, and resource allocation. This combination helps make constraint logic programs expressive and flexible.

Expert systems capture human expertise and use it in the solution process. Since the model employed more closely resembles the real world situation, its solutions could also better fit the context. Gunadhi and his colleague\textsuperscript{16} introduce an automated timetabler that combines data model and knowledge base, developed using object-oriented methodology. The separation of data, knowledge, and algorithms provide flexibility to deal with changes, and the incorporation of human expertise helps in reducing the search space for a feasible solution. Solotorevsky et al.\textsuperscript{19} develop a rule-based language, called RAPS, for specifying resource allocation problems and timetabling problem. The language enables the specification of a problem in terms of resources, activities, allocation rules, and constraints, and thus provides a convenient knowledge acquisition tool. Dhar and Ranganathan\textsuperscript{15} propose the use of an expert system, called PROTEUS, for the allocation of teachers to courses, and compare it to integer programming techniques. For predictive scheduling of passenger trains, Isaai and Cassaigne\textsuperscript{17} and Isaai and Singh\textsuperscript{18} introduce a lookahead, constraint-based algorithm that is designed using an object-oriented approach. In their approach, expert knowledge is used as heuristics for finding practical solutions and combined with the constraint-propagation technique.

Hopfield and Tank\textsuperscript{20} first applied the parallel characteristics of neural networks to solve the optimization problems. The initial results of the Hopfield neural networks are infeasible. Several researchers have modified the approach in an attempt to overcome the limitations of Hopfield neural networks, but the field has never truly recovered from its controversial beginnings. Pellerin and Herault\textsuperscript{21} use neural networks to prevent conflicting assignments as part of a manual scheduling procedure. Smith et al.\textsuperscript{22} propose a discrete Hopfield neural networks that maps the timetabling problem onto a neural network and obtains better and faster results. The extensive computational effort required, however, makes neural networks less impressive.\textsuperscript{32}

3. PROBLEM STATEMENTS OF THE UNIVERSITY TIMETABLING

The timetabling problem is to fix a sequence of meetings between teachers and students in a prefixed periods of time (typically a week), satisfying a set of constraints of various types. Schaerf\textsuperscript{33} classifies the timetabling problems into three main classes: school timetabling, course timetabling, and examination timetabling. The focus of this article is on solving the course timetabling. In general, the university course timetabling problem is to schedule a set of sessions for each
course within a given number of rooms and time periods. More specifically, each instructor offers several courses in each semester and each course has one to five lectures per week depending on the credit hours of the course. Instructors can choose one set of time periods on which they prefer the lecturing to take place. Besides, they can also exclude another set of time periods.

The process of course scheduling begins by proposing courses and time periods. Each instructor fills in a course form that includes the course title, preference time slots, and exclusive time slots. After collecting all the course information offered by instructors, scheduling can then begin. After each course has been scheduled, the department will send the draft timetable to all instructors for the purpose of validation. If there is no problem, the timetable will be sent to the Dean of Academic Affairs. Otherwise, the department will adjust the timetable to satisfy the instructors’ new requirements.

From the problem description above, we can identify three actors including the instructor, the administrator, and the student. Every actor has its own objective:

- For the instructor, the objective is to schedule the courses into the preference time frame.
- For the administrator, the objective is to schedule all the courses into available time periods and classrooms, and to satisfy as many constraints as possible.
- For the student, the objective is that all courses are as disjoint as possible.

Based on the requirements, several constraints can be identified that the courses scheduling system must satisfy:

1. Instructors usually offer more than one course, and these courses should not be scheduled at the same time.
2. The courses cannot be scheduled at the same time as other required courses in the same class.
3. Only one course can be assigned to a classroom at any point of time.
4. Only available time slots of a classroom can be scheduled.
5. The registered students of a course should not exceed the capability of the classroom.
6. The scheduled times of courses should fall within the preference sets and none of the exclusion sets as much as possible.
7. The lectures of a course should not be scheduled in the continual time slots in a week period. The best scheduling policy for a three-credit-hour course is to schedule two lectures in continual time slots and the other one in another isolated time slot on another day.

Constraints can be further classified as hard or soft: the former have to be satisfied, while the latter are desirable but can be relaxed to attain a feasible solution. In our example, constraints one to five are hard constraints, and the constraints six and seven are soft.

A university timetabling problem in the Department of Computer Science and Information Engineering at NCU is used as an illustrate example to demonstrate our approach. The department has 19 faculty members and more than 550 students. The model was tested with real data containing 42 subjects to be scheduled into 40 time slots and 6 rooms.
4. CONCEPTUAL MODELING THROUGH TASK-BASED CONCEPTUAL GRAPHS

In order to automate the software development process, we propose the use of TBCGs\textsuperscript{34} for conceptual modeling and to serve as a basis for transforming specifications into executable codes (see Figure 1). First, we use TBCGs to capture and represent a conceptual model for the problem domain. To construct a conceptual model, task-based specification methodology (TBSM)\textsuperscript{34} 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 automation. Second, the contextual retrieval mechanism\textsuperscript{23} proposes a multiple-viewed approach to help to acquire specifications and elicit information in contexts. To automate the reuse of existing TBCGs, the contextual retrieval mechanism uses fuzzy similarity to automatically match two surrounding contexts of specifications and utilizes purpose-directed analogy for retrieving reusable specifications. Third, the verification of TBCGs\textsuperscript{24} 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 inter-level and intralevel consistency based on operators and rules of inference inherited in conceptual graphs. Finally, we operationalize TBCGs for the execution of task-based specifications. Once TBCGs have been constructed and verified, a blackboard system will automatically transform TBCG specifications into an executable program. Major constituents of a software system such as database schemas, the knowledge base, and an initial user interface can also be generated by transforming TBCGs into their corresponding counterparts.

4.1. 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

![Figure 1. Automated software development processes by TBCG.](image-url)
A general notion of tasks is that 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 that 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 in the next level. The notion of task structure (i.e., task/method/subtask)\textsuperscript{35,36} is adopted for the process refinement.

The conceptual graph\textsuperscript{37,38} 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.\textsuperscript{39} 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.\textsuperscript{40,41} 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 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. Details of task-based specifications in conceptual graphs can be found in Ref. 34.

Referring to our course scheduling example, various components modeled using TBCG are fully discussed below.

Figure 2 shows a task structure of the course scheduler system. We will focus here on the Schedule-An-Instructor task (i.e., $T_2$). The functionality of the task Schedule-An-Instructor is to schedule all the courses that the instructor teaches in
the available time slot and classroom while various requirements and constraints are satisfied. This is achieved by iterating between three subtasks:

1. Select-A-Course ($T_{21}$), which chooses the course among nonscheduled courses that the instructor lectures
2. Schedule-A-Course ($T_{22}$), which attempts to schedule the current course for the current instructor
3. Conflict-Resolution ($T_{23}$), which resolves time conflicts

A correct specification would indicate that Conflict-Resolution should be skipped under the situation that the current course is successfully scheduled for the current instructor. It is also desirable to indicate that if some courses have not been scheduled for the instructor yet, the system should repeat another round to select a course, schedule the course, and resolve conflicts. The specifications in conceptual graphs for the tasks and methods relevant to this example are shown in Figure 3 through 6.

**Domain Model.** Since domain 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
Figure 3. The task Schedule-An-Instructor.

Figure 4. A method of the task Schedule-An-Instructor.
of Schedule-A-Course in Figure 5, the domain model expresses three terms and their attributes; for example, the first term (i.e., Course) has seven attributes (i.e., Name, Session, Class, Required, Credit-Hours, Meeting, and Location). The referent of each concept attaches an "@" to indicate the cardinality of the concept.
**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 as referents 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 Figure 5), in the task specification of Schedule-A-Course, the state model expresses that the state [(current) → [Instructor] → (instruct) → [Course] ← (current)] will be changed into another state [(current) → [Instructor] → (instruct) → [Course] ← (scheduled)] under the conditions that (1) constraints $T_{22} - L_1, \ldots, T_{22} - L_5$ are satisfied, and (2) the task Schedule-A-Course is performed.

**Functional Specifications.** Preconditions, protections, and rigid postconditions are represented by propositions. A soft postcondition can be viewed as a fuzzy proposition with fuzzy concepts and fuzzy conceptual relations. Consider the task Schedule-An-Instructor in Figure 3: The precondition of the task represented by a proposition says that there exists a current instructor who lectures a set of courses. The postcondition of the task expresses that the certain courses lectured by the instructor is scheduled.

**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 retracted its input concepts. 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., $\langle (T) \rangle$) and a START concept (a subtype of STATE) for the beginning state. Second, the convention of viewing a conditional test as a task (e.g., see Ref. 45) is also adopted, denoted as $\beta^?$, 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 $\sim_{\text{final}}$. The difference between selectional and conditional operators is that the expression $E_1$ will not be performed unless the conditional test for $\beta_1^?$ is tested to be true, both of which are not 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 $E$, while the
second demon will be invoked by two input tokens, $\beta$ and $E$, to represent
the notion of iteration condition. The third demon is to indicate the exit condition.
The mapping of TSE operators to conceptual graphs is summarized in Table I.

**Method Specifications.** A method 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.
Refer to TSE of the method in Figure 4, $\langle T \rangle$ and $[START]$ indicate the initiator
demon and the beginning state, respectively. The task $T_{21}$ is performed first while
condition $\beta_2$ is tested to be true, and the state is then changed into $[\varphi(T_{21})]$. Next,
the task $T_{22}$ is performed and the state will be changed into $[\varphi(T_{22})]$. Subsequently,
there exist two possible branches: (1) when the condition $[\beta_3]$ is tested to be
true, the task $T_{23}$ will be invoked; and (2) when the condition $\neg[\beta_3]$ is tested to
be true, the state $[\varphi(T_{22})]$ remains unchanged. In both cases, $[\beta_2]$ will be tested to

<table>
<thead>
<tr>
<th>TSE operators</th>
<th>Syntax</th>
<th>In conceptual graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediately follow</td>
<td>$(T_1, T_2)$</td>
<td>$\langle T \rangle \rightarrow [\varphi(T_1)] \rightarrow [\varphi(T_2)]$</td>
</tr>
<tr>
<td>Follow</td>
<td>$(T_1; T_2)$</td>
<td>$\langle T \rangle \rightarrow [\varphi(T_1)] \rightarrow [\varphi(T_2)]$</td>
</tr>
<tr>
<td>Selectional</td>
<td>$(e_1 \lor e_2)$</td>
<td>$\langle T \rangle \rightarrow [START] \rightarrow [\varphi(e_1)]$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$(\beta_1 e_1 \lor e_2 \beta_2)$</td>
<td>$\langle T \rangle \rightarrow [START] \rightarrow [\varphi(e_1)]$</td>
</tr>
<tr>
<td>Optional</td>
<td>$\epsilon$</td>
<td>$\langle T \rangle \rightarrow [START] \rightarrow [\varphi(\epsilon)]$</td>
</tr>
<tr>
<td>Iteration</td>
<td>$(\beta \epsilon)^*$</td>
<td>$\langle T \rangle \rightarrow [START] \rightarrow [\varphi(\epsilon)]$</td>
</tr>
</tbody>
</table>
decide whether the iteration will be performed again or not. The states attached by (final) indicate the final states.

4.2. Operationalizing Task-Based Conceptual Graphs

Our approach operationalizes TBCGs by providing automatic translation into an expert system in CLIPS. To transform TBCGs into an executable system, it is needed to take multiple constituents of a software system such as database schemas and the knowledge base into consideration. In our approach, the goal of the problem-solving model is to transform a task into its corresponding software components. To achieve this purpose, we adopt the blackboard model (see Figure 7) that consists of a blackboard structure and five knowledge sources with their control components. The problem-solving state data are kept in the blackboard structure that is hierarchically organized into six levels of analysis. Knowledge sources produce changes to the blackboard that lead incrementally to a solution. Interactions among the knowledge sources take place solely through changes on the blackboard. Whenever a piece of knowledge is obtained, the blackboard system will automatically produce executable programs that will lead incrementally to a solution. Accordingly, our system exhibits both an opportunistic and incremental characteristics in dealing with a large amount of knowledge to generate software.

An element in the task level of the blackboard represents a task specification in TBCG. The task partition knowledge source breaks a task down into four models: domain model, state model, functional model, and behavior model, to facilitate the extraction of multiple views. Based on the domain model extracted from the task, the database construction knowledge source can automatically build up the database schemas. The rule extraction knowledge source generates rules from:
(1) the state model, which contains knowledge about constraints, constraints satisfactions, states, and functions, (2) the functional model, which describes precondition, protection, and postcondition, and (3) the behavior model, which characterizes the dynamic behavior of a task. The UI information extraction knowledge source extracts information about formats and structures of knowledge from the domain model, information about human-computer interactions from the state model, and information about the system flow from the behavior model. Finally, in the knowledge base level, the knowledge base synthesis knowledge source integrates all rules from several tasks to form a knowledge base based on the notions of task structures.

4.2.1. Extracting into Database Schemas

Each fact in CLIPS is similar to a tuple in relational database schema and can be defined by the deftemplate construct. The domain model of a task is composed of a network of concepts and conceptual relations. As domain models are similar to entity-relationship (ER) models, the transformation of concepts and conceptual relations to entities, attributes, and relations in ER models is straightforward. The database construction knowledge source first translates a domain model into deftemplate constructs for the purpose of reasoning in CLIPS and then maps these templates into relational database schemas for the purpose of storing facts. Table II indicates the mapping from domain models into deftemplate constructs with one-to-one correspondence. As a deftemplate construct is similar to a relational database schema, we can translate the construct into Oracle database formats directly.

<table>
<thead>
<tr>
<th>Domain model</th>
<th>The deftemplate construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>[C1:<em>] → (R) → [C2:</em>]</td>
<td>(deftemplate C1 (slot C2))</td>
</tr>
<tr>
<td>[C1:<em>] → (R) → [C2:{</em>}]</td>
<td>(deftemplate C1 (multislot C2))</td>
</tr>
<tr>
<td>[C1:<em>] → (R) → [C2:</em>]</td>
<td>(deftemplate C2 (multislot C1))</td>
</tr>
<tr>
<td>[C1:{<em>}] → (R) → [C2:</em>]</td>
<td>(deftemplate C1 (multislot C2))</td>
</tr>
<tr>
<td>[C1:{<em>}] → (R) → [C2:{</em>}]</td>
<td>(deftemplate C2 (multislot C1))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:</em>] → (R2) → [C3:*]@1)</td>
<td>(deftemplate C1 (slot C2) (slot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:</em>] → (R2) → [C3:*]@1)</td>
<td>(deftemplate C1 (slot C2) (multislot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:</em>] → (R2) → [C3:*]@1)</td>
<td>(deftemplate C1 (multislot C2) (slot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:</em>] → (R2) → [C3:*]@n)</td>
<td>(deftemplate C1 (multislot C2) (multislot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:</em>] → (R2) → [C3:*]@n)</td>
<td>(deftemplate C2 (multislot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:{</em>}] → (R2) → [C3:*]@n)</td>
<td>(deftemplate C1 (multislot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:{</em>}] → (R2) → [C3:*]@n)</td>
<td>(deftemplate C2 (multislot C3))</td>
</tr>
<tr>
<td>[C1:<em>] → (R1) → ([C2:{</em>}] → (R2) → [C3:*]@n)</td>
<td>(deftemplate C3 (multislot C2))</td>
</tr>
</tbody>
</table>
To ensure the key constraint and entity integrity constraint will never be violated, we add an id slot to each deftemplate construct. The id slot must be unique and not null for every fact in CLIPS. To avoid the possible violation of referential integrity constraint, the deletion of a fact may cause the deletion of a field in other facts that refer to it.

4.2.2. Extracting into the Knowledge Base

Essentially, functions are computed by operationalizing a network of actors; while state transitions are executed by operationalizing a sequence of demons. To perform functions, we define a set of actor primitives by which composite actors can be represented in terms of these primitive actors. The modularity of an actor is accomplished by translating each actor definition into an independent module in CLIPS. Facts and rules within an actor’s module will not affect those within other modules. To execute state transitions, we also define a set of TSE operators by which complex task sequence can be expressed in terms of these operators. The modularity of a method is accomplished by translating each task into an independent module in CLIPS. The execution of a TSE expression within a task’s module is irrelevant to other tasks.

Transforming Functions into CLIPS. TBCG’s functions are operationalized by automatically translating actors into CLIPS. We have defined a set of actor primitives (see Table III) that can be directly mapped to functions in CLIPS with one-to-one correspondence. These actor primitives provide four groups of primary functions, including test functions, manipulation operators, I/O functions, and mathematical operators. Furthermore, these primitives comprise a substrate on which one can define composite actors.

Our approach allows the set of actor primitives to be extensible; that is, new primitives can be defined in terms of existing actor primitives. We use actor definitions to serve as the mechanism by which one can define the semantics of composite actors. In Sowa’s conceptual graphs, an actor can be defined as combinations of other actors in a dataflow graph. Since actor primitives can be executed by translating them into corresponding codes in CLIPS, a composite actor that is composed of actor primitives can then be executable.

A composite actor is operationalized by means of executing its actor definition. Since an actor definition is comprised of a set of primitives that are directly mapped to CLIPS, a composite actor can then be executed via translating its actor definition into executable codes. For the automatic transformation, we have developed a TBCG CASE tool to create task-based specifications in conceptual graphs and to translate them into executable CLIPS codes.

Transforming State Transitions into CLIPS. TSE operators are used to specify the sequence of subtasks to characterize the behavior of their parent task. Each task is viewed as an independent module in which the internal structure of any task is irrelevant to other tasks. In our approach, a task is translated into a defmodule construct. Since each task forms an individual module in CLIPS, the sequence of subtasks can be executed by using the focus command to choose which task’s module would be executed. As the current focus is the top value of the focus stack,
the focus command pushes a module onto the top of the focus stack and the previous value of the current focus will be remained. When the agenda of the current focus becomes empty (i.e., no rules are fired), the current focus is popped up. Rules outside the current module cannot be fired until the modules they belong to are focused upon.

An operator in the sequencing group, such as follow operator (see Figure 8), is composed of two continuous demons with associated concepts for their input states, input tasks, and output states. Each transition demon can be translated into a defrule construct within the module of its parent task, in which (1) its input state is retracted, (2) its output state is asserted, and (3) its input task is focused upon.

The conditional operator (see Figure 9) is composed of two branching demons with associated concepts including input states, input conditions, input tasks, and output states. Each demon can also be translated into a corresponding defrule construct in which (1) its input condition is matched, (2) its input state is retracted, (3) its output state is asserted, and (4) its input task is focused upon. For example, the first demon in Figure 9 will be translated into two rules as follows (assuming
that condition $\beta_1$ is the conceptual graphs $[Instructor] \leftarrow (scheduled)$ and $\beta_2$ is $(NEG) \rightarrow [[Instructor] \leftarrow (scheduled)]$.

\[
\text{(defrule Module-Name::S-T0-to-S-T1)}
\begin{align*}
\text{?}f1 & \leftarrow (\text{S-T0}) \\
& \text{(exists (Instructor (id ?) (state scheduled) (status ?) (exclusive-set $?$) (preference-set $?$) (course $?$)))} \\
\Rightarrow & \text{(retract ?f1)} \\
& \text{(assert (S-T1))} \\
& \text{(focus T1)}
\end{align*}
\]

Figure 9. The condition operator.
The iteration operator (see Figure 10) is composed of a self-loop transition demon (with its input state, input condition, input task, and output state) and an exit transition demon (with its input state, input condition, and output state). Both the self-loop demon and the exit demon will be translated into defrule constructs. For example, the self-loop demon in Figure 10 will be translated into the following rule (assuming that the iteration condition $\beta_2$ is the conceptual graphs ($NEG \rightarrow [[Instructor] \leftarrow (scheduled)]$).

(defrule Module-Name::S-T2-to-S-T2-iteration
 ?f1 <- (S-T2)
 (not (exists (Instructor (id ?) (state scheduled) (status ?))
 (exclusive-set $?)) (preference-set $?)
 (course $?)))
 =>
 (retract ?f1)
 (assert (S-T2))
 (focus T2))

When the module of a task is focused upon, a fact ($START$) will be asserted if the task’s precondition is satisfied. Subsequently, ($START$) will cause a series of rules (generated by TSE operators) to be fired. The dynamic behavior of TBCGs can then be executed by composing a set of TSE operators.

5. NCU TIMETABLING SYSTEM GENERATED THROUGH TBCG

The university timetabling system at NCU can be divided into two parts. One is the user interface in Java as the front-end; another is the expert system in CLIPS through TBCG specifications as the kernel. We will focus our attention on the latter. The overall architecture of our system is illustrated in Figure 11.

---

Figure 10. The iteration operator.
5.1. Database Schemas

Each connected conceptual graph in the domain model will be mapped to a database schema. Consider the example of the task Schedule-A-Course (see Figure 5). The database construction knowledge source will generate three templates in CLIPS, including Course, Instructor, and Available-Location. The deftemplate construct of Course is shown as follows:

(deftemplate Course (slot id) (slot state) (slot name) (slot session) (slot class) (slot required) (slot credit-hours) (multislot meeting) (multislot location))

As a deftemplate construct is similar to a relational database schema, we can translate the construct into Oracle database formats directly. First, the template name is mapped to a relation table name in relational database schema. Second, each slot is mapped to an attribute column whose type is a built-in data type. Third, a multislot is mapped to an attribute column whose type is a collection type. Finally, facts can then be mapped to tuples in the created object relational database schema directly. Consider the deftemplate construct generated from the Schedule-A-Course task, its corresponding database schema is shown below.

Course

id, state, name, session, class, required, credit-hours, {meeting}, {location}
5.2. Knowledge Base

The rule extraction knowledge source extracts rules for constraints, constraints satisfactions, states, and functions from the state model of each task:

- Each constraint generates two rules in CLIPS: one for testing the satisfaction of a constraint, which is extracted from the rule part of the constraint, and another for repairing the violation of a constraint, which is extracted from the fix-rule part of the constraint. Consider the constraint $T_{22} - L_1$ in Figure 6, the rule part is translated into the rule:

(defrule Schedule-A-Course::T22-L1-rule
  (and (Course (id ?id1) (state current) (name ?name1) (session ?session1)
       (class ?class1)
       (Instructor (id ?id2) (state current) (status ?status) (exclusive-set $?exclusive-set2)
       (preference-set $?preference-set2) (course $?course2)))
  (test (member $?id1 $?course2))
  (forall (Course (id ?id1) (state current) (name ?name1) (session ?session1)
           (class ?class1) (required $?required1) (credit-hours $?credit-hours1) (meeting $?meeting1) (location $?location1))
  (test (not (member $?one1 $?exclusive-set2))))
=>
(assert (constraint-satisfaction (constraint T22-L1))))

- All constraints in a state model form a constraint hierarchy based on the strength. The constraints level of greatest strength denoted as $C_0$ must be satisfied; others are desirable (that is, preferable to be satisfied). A constraint hierarchy is said to be satisfied only when one state can satisfy at least the $C_0$ level. For each level in the constraint hierarchy, one rule is generated for testing if the proper level is satisfied. A current level is satisfied only if all constraints within current level and constraints above current level (i.e., greater strength than current level) are satisfied simultaneously.

- The satisfaction of a constraint hierarchy causes a state in state model to get changed. Hence, each state model will generate one rule for executing the state transition. Consider the state model in Figure 5, the input state will be retracted and the output state will be asserted when its associated constraint hierarchy is satisfied.

(defrule Schedule-A-Course::State-Transition
  ?f1 <- (task-is-satisfied)
  ?f2 <- (Instructor (id ?id2) (state current) (status ?status2) (exclusive-set $?exclusive-set2)
    (preference-set $?preference-set2) (course $?course2))
  ?f3 <- (Course (id ?id3) (state current) (name ?name3) (session ?session3)
    (class ?class3) (required $?required3) (credit-hours $?credit-hours3) (meeting $?meeting3) (location $?location3))
  (test (member $?id3 $?course2))
=>
(retract ?f1 ?f2 ?f3)
(assert (Instructor (id ?id2) (state current) (status ?status2) (exclusive-set $?exclusive-set2)
    (preference-set $?preference-set2) (course $?course2))
(assert (Course (id ?id3) (state scheduled) (name ?name3) (session ?session3)
    (class ?class3) (required $?required3) (credit-hours $?credit-hours3) (meeting $?meeting3) (location $?location3)))


To extract rules for the precondition and the protection from the function model of each task:

- Each precondition in functional models is translated into a rule to test if the precondition holds. Once a task is focused upon, its precondition will be automatically checked to see if the task can be invoked or not. The current focus will be returned (using the \textit{return} function), if the precondition will not hold.
- The protection of a task represents the local invariants and cannot be violated. Hence, a protection is translated into a rule with the highest salience. Whenever a fact is changed, the protection rule will automatically examine the local invariants and restore the previous state if any violation occurs.

Finally, to extract rules about task sequences from the behavior model of each task, task state expressions will be translated into their corresponding rules through demons. Each demon can be transformed into a corresponding \textit{defrule} construct in which its input condition (if any) must be matched, its input state must be retracted, its output state must be asserted, and its input task (if any) must be focused upon.

Once all rules in each task are extracted, the \textit{knowledge base synthesis} knowledge source will integrate these rules using task structures. A task structure is an overall view of the system and can be used to lump all relevant rules together. According to the task structure, the \textit{knowledge base synthesis} knowledge source collects one task at a time from the highest level task. Therefore, we can ensure that rules for each task will not be omitted, and all rules in a system can be integrated under the notion of task structures.

6. CONCLUSION

In this article, we propose an automatic software engineering approach, called TBCGs, to addressing the university timetabling problem. TBCG provides the automation of software development processes, including specification, verification, and automatic programming. A university timetabling system in the Department of Computer Science and Information Engineering at NCU is used as an illustrative example to demonstrate the proposed approach.

Our approach offers several benefits that are useful for addressing the challenges in the timetabling problem:

- Modification can be directly performed on the specifications rather than on the source code; moreover, hard and soft constraints can be easily inserted or removed.
- The separation of facts, knowledge base, and the inference engine provides greater flexibility in maintaining the timetabling system.
- TBCGs 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 task decomposition mechanism provides the capability to deal with large-scale timetabling problems.

Acknowledgments

This research is partially sponsored by the National Science Council (Taiwan) under grant NSC92-2213-E-008-001.
References