ANALOGY SUPPORTS FOR SOFTWARE REUSE AND KNOWLEDGE MANAGEMENT

Chung-Horng Lung[a], Gerald T. Mackulak[b], and Joseph E. Urban[c]
[a]Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada
chlung@sce.carleton.ca

[b]Department of Industrial Engineering
Ariona State Univeristy, Tempe, AZ
mackulak@asu.edu

[c]Department of Computer Science & Engineering
Arizona State Univeristy, Tempe, AZ
joseph.urban@asu.edu

1. INTRODUCTION

Analogical reasoning is critical in problem solving and is fundamental in learning and cognitive development, because analogy is a key factor in hypothesis formation, explanation, and the definition of abstract concepts [1,2,4]. Many analogy theories have been proposed to solve the problem A:B::C:X. Given that A is related to B as C is related to some X, find that X. For example, car:engine::gear:tooth.

Software reuse is similar to analogical reasoning in some respects. To reuse an existing solution for a new problem, we are essentially solving a problem like p(X):s(X)::p(Y):s(X'), where problems p(X) and p(Y) are identical or analogous, and solutions s(X) and s(X') are identical or similar. It may or may not be easy to identify the similarities and differences between p(X) and p(Y). Software reuse can also be at different levels, such as software architectures or at the code level.

Knowledge management deals with the way of how to capture, organize, update, share, and use the knowledge. Knowledge management is closely coupled with analogy in that both areas discuss the transfer of knowledge from one problem (base) to another problem (target).

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35602-0_35

These three areas have many in common. All deals with similar dimensions, including representation, retrieval, elaboration, mapping, evaluation, integration, generalization, classification, and learning. Furthermore, analogy community advocates that rich representation must include syntactic, semantic, and pragmatic components [3,4]. Same ideas are used in software engineering discipline.

The objective of this paper is to present some crucial discoveries in analogy [1-4]. We have developed an approach based on analogy to support software modelling. A study in the manufacturing problem domain is briefly described. The study shows reuse from the discrete manufacturing domain to continuous manufacturing problem area. Compared with the base, the development time for the target problem artifacts is significantly reduced from days to hours. The result suggests researchers in software engineering and knowledge management may benefit more by exploring further in the analogy community.

2. ANALOGICAL REASONING

Here, we highlight two critical aspects of analogy, namely modelling of high-order relations and reasoning of semantic relations.

A often discussed area in various analogical reasoning theories is relations [1-4]. Relations between concepts, not just the representation of individual concepts, are the key in identifying analogy. Relational modelling in this paper emphasizes two aspects: higher-order relations and classification of semantic relations.

The central idea in Gentner's structure-mapping theory [3] is the principle of systematicity, which states that analogy is a mapping of systems of relations governed by higher-order relations with inferential import, rather than isolated predicates. Higher-order relations capture the relation of relations.

Take the analogy between the structure of our solar system and the atom system as an example. Several relations between the sun and a planet include distance, attract, more-massive, revolve-around, and hotter-than. Together, distance, attract, more-massive, and revolve-around form a higher-order relation:

CAUSE [distance(sun, planet), attract(sun, planet), more massive(sun, planet), revolve-around(planet, sun)]

This higher-order relation can be mapped to the atom system. Some solutions for the solar system can then be mapped to the atom system. Isolated relations, such as the sun is hotter-than the planet, are discarded in the mapping phase.

Next, the ability to readily compare relations means that relations are readily decomposed into more primitive elements [2]. People readily compare relations. This requires that relations can be decomposed into aspects in which they are the same and aspects in which they differ. Bejar et al. [1] presented a taxonomy of the semantic relations. Two of relation classes are closely related to object-oriented methods. They are class inclusion (Is-A) and part-whole (Has-A).

3. APPLICATION OF ANALOGY TO SOFTWARE REUSE AND KNOWLEDGE TRANSFER

The idea of software reuse through analogy is not new [5,6]. In order to support the modelling of higher-order relations, the approach comprises object modelling, functional modelling, relational modelling, and dynamic modelling. Object, functional, and dynamic modelling are similar to object-oriented methods. Specifically, the main artifacts adopted from those modelling phases include entity relationship diagram, data flow diagram, functional descriptions, and a rule-based Petri net representation. The artifacts can be replaced with other similar representation schemes.

Object Modelling. We start with a detailed comparison of the main components between the discrete and continuous manufacturing. The main purpose is to identify both similarities and differences. Both domains have similar material handling systems, whose primary operation is to move and store materials, parts, and products. In a continuous problem, the number of machines is fewer, but generally each machine is costly and performs more complex operations. Because the stations are more sophisticated in the continuous domain, planning for maintenance and equipment failure is more important than in the discrete domain. Another significant difference is the scheduling process due to the difference in product type and machine stations. As a result, simulation models needed for these two domains are also different. In the discrete domain, finite state machines and discrete event models are commonly used. In the continuous domain, difference equations or differential equations are needed.

Based on above comparison and observations, and the object model for discrete manufacturing, an entity relationship diagram (ERD) for the continuous domain is derived. Based on the analysis, entities queue and sensor, and their associated relationships are removed for the target domain.

Functional Modelling. The main artifact for the functional model is data flow diagrams (DFD). The DFD derived for the target share many similarities with that of the base. There are eight main processes in the base problem. Seven processes are derived from the discrete manufacturing and one (dealing with the queue) is removed. Five processes are repeated use without modifications. Two processes that deal with monitoring of product position is modified to deliver/remove materials and monitoring the product quality, respectively.

Dynamic modelling. Dynamic modelling includes modelling of high-order relations. Dynamic models capture more detailed and more specific information. Modelling of higher-order relations, on the other hand, conveys high-level cause-effect information. Currently, there is no appropriate technique or formal mechanism for modelling higher-order relations, i.e., relations of relations. In traditional OO modelling, we usually capture relations between classes or objects. Those types of relations represent low-order relations.

The idea is similar to the dynamic modelling in OO modelling languages like UML. Features in UML to model behavioural diagrams, e.g., use case diagram, sequence diagram, and collaboration diagram, represent collection of relations, which is conceptually similar to the structure of relations proposed in analogical reasoning. However, many OO applications focus on the solution space. To be effective, the problem space should be explicitly and clearly modelled. Moreover, we also need to represent the problem space beyond objects and lower-order relations. Improvements in representation will facilitate the identification of similarities and differences between p(X) and p(Y) as stated in the introduction.

Relational Modelling. Class inclusion (Is-A) and part-whole (Has-A) relations proposed in analogy are also widely used in OO modelling. Bejar et al. [1] listed five members for the Is-A class and ten members for the Has-A class, which might worth further exploring.

For this study, part-whole relations are examined in more detail. For instance, an engine is part of a car. Specifically, car:engine is in the Object:Component category [1]. However, there are differences beyond the part-whole level. For the continuous manufacturing, there also exist part-whole relations, but they fall into the Mass:Portion category as in the milk:skim milk example. The distinction between the two problems directs us to do further investigation. As a result, two different types of machine stations and control processes are identified for these two domains, albeit these two domains share a similar higher-order relation.

Part-whole relationships are widely used in OO modeling. There is confusion about the relationship [7]. In UML, composition is advocated as a special form of aggregation within which the parts are inseparable from the whole. The idea of separating composition from aggregation is similar to the classification of semantic relations. However, there exist more differences than just aggregation and composition. For example, *car:engine* and *department:company* are different even though they both share part-whole relationships. In the first case, car has one engine and only one. Other parts, e.g., transmission, are very different from engine. However, departments within a company share many similarities. In [1], both examples share part-whole relationship, but *car:engine* is classified as *Object:Component* and *company:department* belongs to *Collection:Member.*

4. CONCLUSION AND FUTURE DIRECTIONS

This article presented an analogy-based approach to support software ruse and knowledge transfer. A case study was briefly presented. We highlighted lessons that we can learn from analogy. Some important lessons include modelling of higher-order relations, comparison of semantic relations, and OO modelling. Software engineering is a people- and knowledge-intensive business that would benefit from the reuse of past experience. Other relevant methods reported in analogical reasoning could also be applied to software engineering.

5. REFERENCES

1. I.I. Bejar, R. Chaffin, S. Embretson, *Cognitive and Psychometric analysis of analogical problem solving*, Springer-Verlag, 1991.
2. R. Chaffin and D. Herrmann, "Relation Element Theory: a New Account of the Representation and Processing of Semantic Relations", *Memory and Learning: The Ebbinghaus Centennial Conf.*, 1987, pp. 221-245.
3. D. Gentner, "Structure-Mapping: a Theoretical Framework for Analogy", *Cognitive Science*, vol. 7, no. 2, 1983, pp. 155-170.
4. D. Helman, ed., *Analogical Reasoning*, Kluwer Academic Publishers, 1988.
5. C.-H. Lung and J.E. Urban, "An Expanded View of Domain Modeling for Software Analogy", *Proc. of Int'l Computer Software & Applications Conf*, 1995, pp. 77-82.
6. N.A.M. Maiden and A.G. Sutcliffe, "Requirements Engineering by Example: an Empirical Study", *Proc. of IEEE Int'l Symp. on Reqt. Eng.*, 1993, pp. 104-111.
7. A.L. Opdahl, et al. "Ontological Analysis of Whole-Part Relationships in OO-Models", *Info and Software Technology*, 43, 2001, pp. 387-399.