INFORMATICA, 1995, Vol. 6, No. 4, 483–496

THE ROLE OF ANALOGY IN REUSE

Audronė LUPEIKIENĖ

Institute of Mathematics and Informatics
2600 Vilnius, Akademijos St. 4, Lithuania

Abstract. The development of software systems is intensive person–oriented process. Therefore it is essential to use previous experience and knowledge. The extension of reuse through analogy is analysed in this paper. The proposed grouping of analogical methods, rules, etc., relies on their adaptability to application software development. The emphasis here is on analogue as a criterion to safeguard the desirable properties of application software. Knowledge kinds and mechanisms to enable reuse through analogy are discussed.

Key words: reusability, analogy, requirement specification, consistency, completeness, adequacy.

1. Introduction. The possibility of learning to use previous experience and artefacts is very important in software systems development. Working out new problems people usually do not look for a new solution but a similar problem have been solved earlier is looked for. A new problem is often solved by adapting the solution of the old one. Reuse in software systems development encompasses all the resources produced during earlier development. Reuse is usually applied during the software systems implementation phase to reuse code fragments and skills. Reuse during the first software systems development phase has recently begun to be investigated (Horowitz and Munson, 1987; Maiden, 1991).

Reuse is typically analysed as procedure of correctness–preserving nature (De Antonellis and Pernici, 1995; Balzer, 1985). Reuse may benefit from weakening this property and introducing heuristic rules such as analogical reasoning especially when we deal with informal user's specification. Analogical reasoning has been studied by many researchers in relation to problem solving (Carbonell, 1983) and artificial intelligence problems (for example, learning) (Winston, 1982; Gentner, 1983; Greiner, 1988; Hall, 1989). The idea to use analogy in automatic synthesis of programs (code) has been developed in Dershowitz (1985). Analogy as one of the means of reusing specifications during the early stages of software systems development has been analysed in Maiden (1991), Lupeikienė and Lempertas (1990).

Software reuse implies not only the software productivity but also the software quality. Product quality depends on the quality of its components. Quality is possible if its criterion is defined. Analogue can serve as external criterion to safeguard the correctness of application software (Lupeikienė and Lempertas, 1990) especially in the first development stage when we deal with informality.

In this paper the attempts to motivate reuse extension through analogy are presented. Different analogical methods are categorized on their adaptability to application software development. At the end of this paper knowledge kinds and mechanisms to enable reuse through analogy are discussed.

2. Development of application software. People always use their experience for solving new problems. When a new problem is to be solved, a similar problem from the earlier experience is looked for. To use something again means to reuse it. According to Freeman (1987) reusability is an activity that produces a system by reusing something from previous development effort. Knowledge and artefacts are reused in application software development.

Dealing with reusability two general classes of software systems development approaches can be distinguished:

1. Reusable processors, where interpreters for executable high–level specifications are reused (for example, Horowitz and Munson (1987)).
2. Reusable mapping systems, where development of software system is the sequence of transitions from i level description D_i to $i+1$ level description D_{i+1} obtaining executable program/code. The mapping types are:
   - transformation (for example, Balzer (1985), Horowitz and Munson (1987)),
   - translation,
   - refinement (for example, Wirth (1971), Miriyala and Harandi (1989)).

The mapping system can be defined as the pair $\langle D_i, MR_{D_i \to D_{i+1}} \rangle$, where D_i is problem description in i level language, $MR_{D_i \to D_{i+1}}$ – mapping rules from description D_i to D_{i+1}.

Let us take a closer look at transformational systems. Development in the context of transformational view is:

- specification of a new problem in the language of i level obtaining i level description D_i,
- transformation $D_i \rightarrow D_{i+1}$ of i level description into $i+1$ level description, and n level description meets implementation conditions: is compilable, is in target language, is efficient, etc.

Transformation rules are formal and ensure correctness of the result $i+1$ level descriptions. These rules also preserve properties of i level descriptions in $i+1$ level: consistency, completeness, adequacy.

Let us look at the process of application software development. Fig. 1 shows application software life cycle for mapping systems. When a new problem is defined its informal specification is presented. This specification contains information

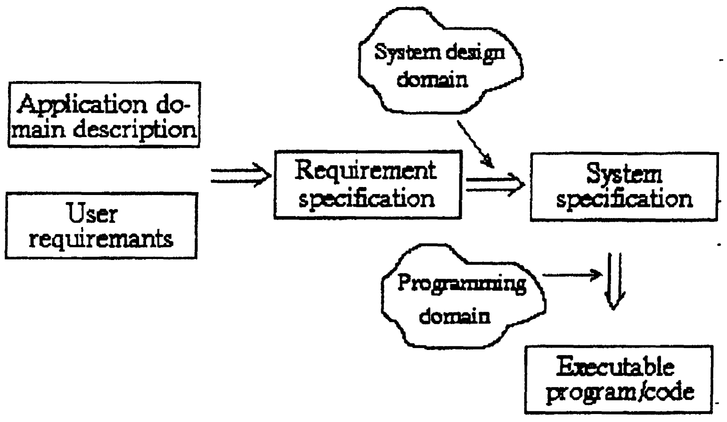

Fig. 1. Software life cycle in mapping systems.

about application domain and description of the task to be solved. The informal specification presented by the user is the background for obtaining requirement specification – a formal high level description reflecting the user's point of view on the system. Requirement specification is mapped into system specification – lower level formal description reflecting system developer's point of view on the system. The last and the best analysed step in application software development is the implementation of system specification obtaining executable code. Two additional domains are inseparable from this process: domain of system design and domain of programming.

All mapping steps are the same (see Lupeikienė and Lempertas, 1990) in the context of solutions of the problems outlined in this paper. Thus below we examine the mapping: application domain description + user requirements $\rightarrow$ requirement specification.

Transformational approach ensures correctness of the resulting executable program/code by construction. It means that requirement specification must be correct, i.e., requirement specification must be consistent, complete and adequate. Problems arise because of mismatches between what the user wrote and what he had intended. The user's information on application domain and tasks to be solved as a rule is:

- insufficient (forgotten or unknown for the user),
- incorrect,
- unverified,
- ambiguous,
- old.

The user omits some information considering it to be well known from experience or common sense.

Reusability implies automation. Standard automatic application software development methods require consistent, complete and adequate requirement specification in a certain formal language. Automation in transformational systems requires consistent, complete and adequate 1 level description D_1 and transformational rules.

There are different approaches proposing methods and tools to obtain consistent, complete and adequate requirement specification. We shall discuss approaches which:

- analyse the specification itself,
- use external knowledge for analysis.

An example of internal analysis of specification is in Balzer (1985). R.Balzer to partially overcome the problem of safeguarding the features of requirement specification proposes to build paraphraser and examine the behaviour itself. The paraphraser improves the readability of specification. Among the classes of tools for dynamic specification analysis: theorem proover, interpreter and symbolic evaluation – the latter is used. These tools allow to find only part of inconsistencies and incompleteness: as much as the user himself can determine from the other information representation form and from consequences of the behaviour generated for the test case.

An example of first steps to use external knowledge for specification analysis is in Levene and Mullery (1982). In order to ensure necessary properties of requirement specification the attempt is made to expand the boundaries of one person's understanding. The acquired requirement specification is compared to a predefined standard one in a formal language. However, in such a way only predefined inconsistencies can be determined and predefined knowledge can be used to supplement requirement specification.

Analogical approach overcomes the enumerated problems and extends the scope of external knowledge being dealt with (Lupeikienė and Lempertas, 1990) for the reason that analogy supports reuse across domains. In the next section we propose grouping of analogical methods relying on their impact to application software development process.

3. Analogy in application software development. Analogy is similarity of a certain type. According to G.Polya (1954) analogy is such similarity which can be expressed on a concept level. Analogical situations correspond to each other by certain relations. Analogical approach is based on presumption that if two situations or descriptions are analogous in a certain aspect they should be analogous in other aspects as well. Such pragmatic view allows to analyse analogy as the mapping between two domains: source and target. Analogy system can be defined as the pair $\langle S, MR_{S\to T}\rangle$, where S is source description, $MR_{S\to T}$ – mapping rules from source to target. From this point of view analogy system is mapping system, where mapping type is analogical derivation and $i = 1$.

Reuse may benefit from weakening formal correctness–preserving nature of transformation systems and/or combining pure mapping types in one system. Analogy has advantages in that it enables a wider variety of previous knowledge and artefacts for reuse than less powerful techniques: matching on similarity, abstraction, classification.

The analogical methods, rules, etc., proposed in Carbonell (1983), Chouraqui (1985), Dershowitz (1985), Gentner (1983), Greiner (1988), Hall (1989), Jantke (1985), Maiden (1991), Miriyala and Harandi (1989), Winston (1982) can be differently categorized. The categorization may rely on reusable knowledge kinds, nature of reusable knowledge (Mili *et al.*, 1995), etc. The categorization relying on the analogy adaptability and impact to application software development is analysed in this paper.

Problems which can be solved by extending transformation systems with analogical reasoning to obtain requirement specification are the following:

A. Requirement specification complement in the case of missing information.

B. Requirement specification consistency, sufficient completeness and adequacy analysis and assurance.

Dealing with the A problem analogical methods can be categorized in two types based on the extent to which the knowledge from analogues are reused:

- methods which aim to map maximum possible knowledge from analogue to target,
- methods which aim to map useful (for concrete problem solving) knowledge.

Methods described in Winston (1982), Gentner (1983) can be mentioned as examples of maximum mapping. They vary in what is thought to be essential to map from analogue to target. The paper Winston (1982) points to the importance of causal relations. The method presented in Gentner (1983) aims to maximize overlapping of relational structures. Mapping rules include the principle of systematicity which considers interconnected relations more important than isolated ones. These two approaches can be applied simultaneously in order to maximize information mapping. However, generally the different analogical methods can be incompatible.

Greiner's (1988) analogical method is an example of usefulness in analogy application. The solving of initial problem in analogical domain proposes the conjectures to be added to the initial domain. Thus the theory, formed by adding the new conjecture, must be able to solve the initial problem too.

In the requirement specification complement, analogical methods which maximize possible mapping knowledge are preferred.

The essence of the B problem is different from the first one. The analogue is suggested to be used as criterion to ensure desirable characteristics of requirement specification. Requirement specification quality improves if qualitative components are reused and quality is possible if its criterion is established.

Reuse is closely tied with adaptation because artefact as a whole or its components cannot be used frequently as they are. Changes can be planned using various parameterization and abstraction techniques. Analogical approach implies poorly planned and unassumed changes, so modification is required to accommodate the needs of a new system.

Modification is the sequence of transitions of i level description from j state to $j+1$ state. The modification rules $MdR_{D_i^j \to D_i^{j+1}}$ signify that changes affect states of i level description and not the level of description language. Depending on the object to be modified and the purpose of modification, we distinguish three approaches (denoted by B1, B2, B3) to software system development.

B1. The users' information about application domains and tasks to be solved $US_1, \ldots, US_m$ is stored with corresponding requirement specifications $RS_1, \ldots, RS_m$. Components in the solution space RS_i can be traced to components in the initial space US_i. The new requirement specification RS is obtained by first matching user's specification US to the known $US_1, \ldots, US_m$ to find analogous user's specification US_i^A. The determined all possible analogies are used to modify analogical requirement specification RS_i^A in order to construct the target requirement specification RS (Fig. 2).

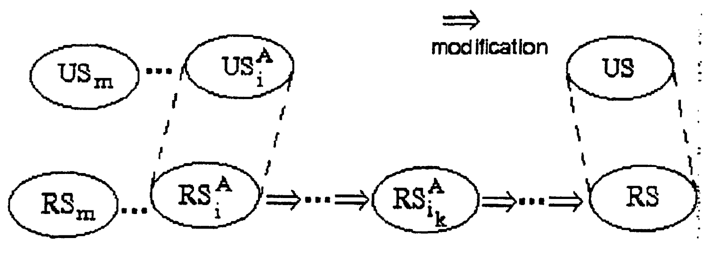

Fig. 2. Specification derivation by analogue modification.

The characteristic example of this approach is presented in Carbonell (1983) where problem solution is derived in the second-order space. The states of the second-order problem space encapsulate analogical solutions or their modifications.

This approach is preferred when the mapping from i level description to $i+1$ level description cannot be characterized by mapping rules $MR_{D_i \to D_{i+1}}$.

Due to the nature of analogy process the derived description D_{i+1} may be incorrect. The additional heuristic rules usually are developed to correct the errors. To the extent as analogy application is correct the desirable properties of i level description D_i are preserved or developed in $i+1$ level description D_{i+1}.

B2. The previous users' specifications $US_1, \dots, US_m$ are stored in the system together with derivation paths $DP_1, \dots, DP_m$ of corresponding requirement specifications. The derivation path is understood as a set of derivation rules together with their application order. Derivation path retrieval is achieved by matching the new user's description US to each $US_1, \dots, US_m$ to select analogical specification US_i^A. The derivation path of analogical requirement specification is modified and used to derive a target requirement specification RS. This approach is shown in Fig. 3.

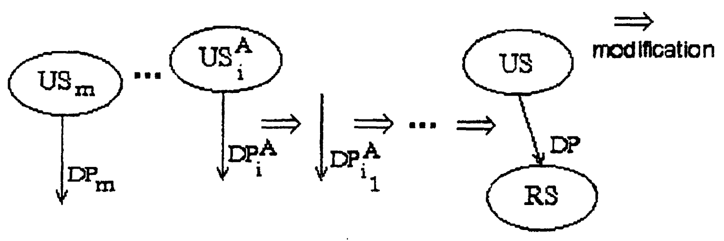

Fig. 3. Specification derivation by analogous mapping paths.

The Greiner's (1988) analogical method can serve as an example of this approach. The author introduces abstractions for organizing clusters of related derivation paths. The derivation of target requirement specification RS is defined by set of heuristics. Heuristics prune and order abstractions–candidates and their instantiations.

This approach is preferred when we cannot store or define all the mapping rules $MR_{D_i \to D_{i+1}}$ and only their useful subset is sufficient to obtain $i+1$ level description. (Useful subset of mapping rules is a collection of necessary mapping rules to derive requirement specification.)

B3. The third approach (Fig. 4) is closest to the traditional transformational approach to application software development. Analogue as the criterion to ensure necessary properties of requirement specification is mostly obvious in this approach.

Previous users' specifications $US_1, \dots, US_m$ and corresponding requirement specifications $RS_1, \dots, RS_m$ are stored in the system. The base require-

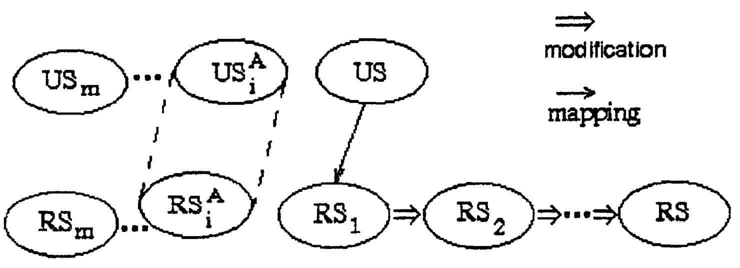

Fig. 4. Specification derivation by modification of base specification.

ment specification RS_1 is obtained from user's specification US. Specification RS_1 rarely meets the consistency, completeness, adequacy conditions. To overcome that problem specification RS_1 is modified using inferred analogies between RS_i^A and RS_1. The prerequisite for exploiting the criterion RS_i^A is the existence of analogy between US and US_i^A. It should be noted that mapping from US to RS_1 is carried out by rules which may include not only correctness–preserving transformations but analogical reasoning rules as well.

Primary results of this approach are presented in Lupeikienė (1992), but more detailed development requires further analysis.

The next section deals with the reusable knowledge kinds to ensure creation of base and target requirement specifications.

4. Knowledge kinds and mechanisms in reusable development. Reuse in application software includes all the knowledge used and produced during development process. Different researches propose different classification of reusable knowledge Freeman (1987), Mili *et al.* (1995). In this paper four reusable knowledge kinds are distinguished to enable both mapping and modification processes in application software development:

- knowledge about domains,
- knowledge about analogues,
- base concepts and rules,
- concepts' base semantics.

This knowledge can be manipulated consistently by:

- generalization–specialization mechanism,
- mechanism of similarity–difference analysis,
- analogical mechanism (Fig.5).

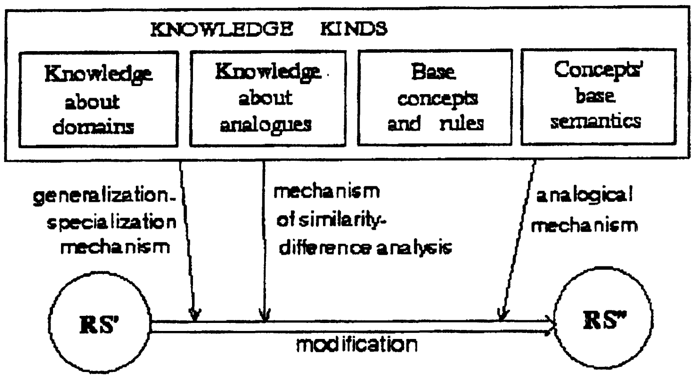

Fig. 5. Knowledge kinds and mechanisms to enable specification modification.

Knowledge about domains is domain-specific, and domains we deal with in development process are different (see Fig. 1). Application domain knowledge is whatever known about the area which problems computer-based system solves. For example, tool-making shop management, mathematics, etc. This knowledge is more stable, less changing than user's requirements and can be used to build not the only application software. System design domain and programming domain knowledge relates to the process of application system development. This may include life cycle models, system architecture models, definitions of data types, definitions of programming language constructs, test plans, etc. Viewed abstractly, $i + 1$ level domain predetermine the $i + 1$ level language to which concepts and constructs i level description is mapped. The development processes may differ in number of domains.

The knowledge about analogues consists of a set of domain descriptions. Not all analogous domains are closely related. Descriptions of the diverse domains are relevant and of great value as well.

Base concepts are general domain-independent knowledge. Base rules include laws, axioms, principles. Base rules imply the possibility to solve general problems or use general means of solutions. They relate the problem at hand with typical solution of this problem. As the example of base rule modus ponens rule can serve, in this case applied not to propositions but to objects and types of relations.

Concept's base semantic reveals the underlying principle or the essence of the concept. The discovery of the underlying principle to a certain extent explains the human understanding of the concept. The underlying principle enables someone to perceive the reason why the instance is a member of the concept and recognize the basis relating members to the concept. Matching on underlying principle is more powerful technique than matching on similarity, on analogy or classification. The first steps toward finding out the technique by which a machine could arrive at concept base semantic were done in Pursvani and Rendel (1987). The development of this proposal requires further analysis and research. It should be noted that real world objects are characterized by structure. The ability to handle structured objects is inevitable to discover the essence of the object.

Two steps in analogical reasoning: analogue detection and analogy application, i.e., mapping from source to target, may be described and controlled by analogical mechanism. Analogical mechanism facilitates the definition of concepts' base semantics as well.

Knowledge is organized into a structured taxonomy. It means that knowledge is linked together by the relation of generalization. Generalization-specification mechanism is needed in analogy application step, in tailoring base concepts and rules to the needs of concrete problem.

The mechanism of similarity–difference analysis is needed to define concepts' base semantics. The influence of different types of similarity (parameteric, incremental, etc.) on discovery of concept underlying principle is outside the scope of this paper. It should be noted that similarity–difference mechanism plays important role in requirement specification creation when initial description is provided by means of differences from some analogue.

5. Conclusion. Reuse is essential in human activity thus the application software development is not exception. The extension of reuse through analogy in application software development was analysed in this paper. The main issues pointed out in this paper are:

- transformational approach to application systems development benefit from complement with analogical reasoning,
- different analogical methods, rules, etc. can be grouped according to their adaptability to application software development into three groups, which use previous artefacts, extend mapping rules and extend modification rules,

- the desirable properties of domain descriptions, specifications, programs, etc., are assured using analogue as a criterion,
- knowledge about domains, analogues, base concepts and rules, concepts' base semantics are to be used to complement the target specification.

REFERENCES

Balzer, R. (1985). A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering,* **SE–11**(11), 1257–1277.

Carbonell, J.G. (1983). Learning by analogy: formulating and generalizing plans from past experience. In R.S.Michalski, J.G.Carbonell and T.M.Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach.* Tioga, Palo Alto, CA pp. 137–162.

Chouraqui, E. (1985). Construction of a model for reasoning by analogy. In L. Steels, J.A. Campbell (Eds.), *Progress in Artificial Intelligence.* England. pp. 169–183.

De Antonellis, V., and B.Pernici (1995). Reusing specifications through refinements levels. *Data & Knowledge Engineering,* **15**(2), 109–133.

Dershowitz, N. (1985). Program abstraction and instantiation. *ACM Transactions on Programming Languages and Systems,* **7**(3), 446–447.

Freeman, P. (1987). Reusable software engineering: concepts and research directions. In P.Freeman (Ed.), *Tutorial: Software Reusability.* IEEE Computer Society Press. pp. 10–23.

Gentner, D. (1983). Structure mapping: a theoretical framework for analogy. *Cognitive Science,* **7**(2), 155–170.

Greiner, R. (1988). Learning by understanding analogies. *Artificial Intelligence,* **35**(1), 81–125.

Hall, R.P. (1989). Computational approaches to analogical reasoning. *Artificial Intelligence,* **39**(1), 39–120.

Horowitz, E, and J.B. Munson (1987). An expansive view of reusable software. In P.Freeman (Ed.), *Tutorial: Software Reusability.* IEEE Computer Society Press. pp. 39–49.

Jantke, K.P. (1985). Program synthesis by analogy – a two–phased approach. In W. Bibel and B. Petkoff (Eds.), *Artificial Intelligence: Methodology, Systems, Applications.* North–Holland, pp. 67–75.

Levene A.A., and G.P.Mullery (1982). An investigation of requirement specification languages: theory and practice. *Computer,* May, 50–59.

Lupeikienė, A., and D.Lempertas (1990). Some remarks on analogical approach to automatic software development. II, 2541–Li. 1–10.

Lupeikienė, A. (1992). Analogical approach for implementation of inheritance and delegation. *Avtomatizatsija protsesov planirovanija i upravlenija,* **13**, 79–89 (in Russian).

Maiden, N.A.M. (1991). Analogy as paradigm for specification reuse. *Software Engineering Journal,* **6**(1), 3–15.

Mili, H., F. Mili and A. Mili (1995). Reusing software: issues and research directions. *IEEE Transactions on Software Engineering,* **21**(6), 528–561.

Miriyala, K., and M.T. Harandi (1989). Analogical approach to specification derivation. *ACM Software Engineering Notes,* **14**(3), 203–218.

Polya, G. (1954). *Mathematics and Plausible Reasoning.* Princeton University Press, Princeton, NJ. 472 pp.

Pursvani, K., and L. Rendel (1987). A reasoning-based approach to machine learning. *Computational Intelligence,* **3**(4), 351–366.

Winston, P.H. (1982). Learning new principles from precedents and exercises. *Artificial Intelligence,* **19**(3), 321–350.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM,* **14**(4), 221–227.

Received November 1995

A.Lupeikienė is a researcher at the Management Systems Department of the Institute of Mathematics and Informatics, Vilnius, Lithuania. Her research interests include conceptual models, methodologies and tools for databases and intelligent information systems design.

ANALOGIJOS VAIDMUO PAKARTOTINAME NAUDOJIME

Audronė LUPEIKIENĖ

Susidūrę su nauja problema, žmonės paprastai ieško ne naujo sprendimo, o panašios, anksčiau spręstos problemos. Žinių, patyrimo, gautų rezultatų pakartotinas naudojimas ne išimtis ir programų sistemų kūrime. Šiame darbe nagrinėjamas pakartotino naudojimo galimybių išplėtimas analogijos metodu. Pasiūlytas analogijos metodų sugrupavimas pagal galimybę juos pritaikyti programų sistemų kūrime ir taikymo tikslą. Pabrėžiamas analogo, kaip kriterijaus, vaidmuo reikiamų programų sistemų savybių užtikrinimui. Išskirti žinių tipai ir mechanizmai, įgalinantys atvaizdavimo ir modifikavimo procesus programų sistemų kūrime papildyti analogija.