Requirements Eng (2007) 12: 1–22
DOI 10.1007/s00766-006-0035-8

ORIGINAL RESEARCH

**Chung-Horng Lung · Joseph E. Urban
Gerald T. Mackulak**

Analogy-based domain analysis approach to software reuse

Received: 19 August 2005 / Accepted: 29 March 2006 / Published online: 5 May 2006

Abstract Domain analysis is an expansion of conventional requirements analysis. Domain analysis can support effective software reuse. However, domain analysis is time consuming and is limited to a particular application area. Analogical approaches to software reuse, on the other hand, often occur across domains. Analogical problem solving is a process of transferring knowledge from a well-understood base domain to a new target problem area. Analogy can facilitate software reuse for poorly understood problems or new application areas. Analogy shares similar concepts with reuse and some analogy theories have been applied to software reuse. However, current research on software analogy often overlooks the importance of analysis for the base domain and does not consider some critical aspects of analogy concepts. Reuse must be based on high quality artifacts, especially reuse across domains. This paper presents an approach to integrate domain analysis and analogy methods. In our view, domain analysis and software analogy have complementary roles. Domain analysis is regarded as a process to identify and supply necessary information for analogical transfer. Software analogy can provide the analyst with similar problems and solutions to reuse previous domain analysis knowledge or artifacts for a new domain. This paper presents case studies to demonstrate the increase of efficiency in applying the approach. Evaluation of the approach from various perspectives is also reported.

Keywords Requirements engineering · Domain analysis · Software analogy · Software reuse

C.-H. Lung (✉)
Department of Systems and Computer Engineering,
Carleton University, K1S 5B6 Ottawa, ON, Canada
E-mail: chlung@sce.carleton.ca
Tel.: +1-613-5202600
Fax: +1-613-5205727

J. E. Urban
Department of Computer Science and Engineering,
Arizona State University, Tempe, AZ 85287-5406, USA
E-mail: joseph.urban@asu.edu
Tel.: +1-480-9653374

G. T. Mackulak
Department of Industrial Engineering,
Arizona State University, Tempe, AZ 85287-5906, USA
E-mail: mackulak@asu.edu
Tel.: +1-480-9656094

1 Introduction

Software reuse has been explored extensively since the late 1980s because of the potential benefits that reuse may bring to productivity, quality, time-to-market, and cost. Reuse of high-quality components is the most influential factor for increasing productivity (Jones 2000). Many actively pursued research areas today, such as design patterns, generative programming, component engineering, and software product lines, have evolved from or are closely tied to reusability. As such, reuse can happen in various forms. Table 1 presents a taxonomy that views reuse from eight perspectives or facets.

Domain analysis is a systematic approach designed for reuse, which can effectively support large scale, vertical software reuse as depicted in Table 1. Domain analysis is comparable to, but expands on the scope of conventional requirements analysis by examining features in a family of systems in order to identify commonalities and variabilities (Prieto-Diaz and Arango 1991). Domain analysis facilitates the construction of flexible software products (typically, artifacts and components) that can be reused (black-box) or tailored (white-box) for similar applications either from top-down or bottom-up in the same problem domain. In other words, domain analysis can support higher levels of reuse, e.g., design, architectures, and domain models (Neighbors 1992).

Another area that has the potential to support reuse is called analogy. Analogy involves transferring *knowledge* (concepts, artifacts, or processes) from past base problem solving experiences to a new problem that shares significant aspects with the past problem and

Table 1 Facets of reuse (adapted from Prieto-Diaz 1993)

By role	By substance	By scope	By mode	By technique	By intention	By products	By size
Design for reuse	Knowledge, ideas, and concepts	Vertical	Unplanned: ad-hoc, opportunistic	Bottom-up compositional	Black-box, as is	Source code	Small scale
Design by reuse	Artifacts and Components	Horizontal	Planned: systematic	Top-down generative	White-box, modified	Design	Large scale
	Processes and procedures			Hybrid		Specifications	
						Objects	
						Text	
						Architectures	
						Domain models	

subsequently constructing solutions (white-box) for the new target problem with the transferred knowledge. In other words, analogy supports design by reusing existing solutions horizontally across domains. Analogy is useful in problem solving and is fundamental to learning and cognitive development, because analogy is a key factor in hypothesis formation, explanation, and the definition of abstract concepts (Hoffman 1995). In problem solving, we need a sense of the whole before analyzing the parts. We can gain such a sense by examining similar or analogous problems.

Analogy can support a higher-level of reuse than domain analysis. However, analogy is based on an assumption that the base problem is well-understood and there are similarities between the base and the target, which often occur in an opportunistic manner depending on the nature of the problems. Domain analysis can help better understand a problem area and hence support the assumption usually made in analogy. Domain analysis also increases the opportunity to identify *good* analogy. Domain analysis manages the identification, capture, and evolution of application or domain-specific knowledge and reusable information in order to promote systematic reuse in an application area. On the other hand, analogy often occurs across domains. Many problems may be different syntactically, and yet these problems share a similar knowledge structure. Reuse of problem solving experience and knowledge from one domain to another is the potential of analogy, especially for a complex domain and a sophisticated task like domain analysis.

Thus, domain analysis and analogy can have complementary roles. Domain analysis, if properly conducted, can supply quality information which is useful for analogy identification and mapping to a target domain. Conversely, analogy can facilitate future domain analysis, particularly for new application areas or poorly understood problems by transferring knowledge and experience from an existing area. However, to support analogy transfer, we first need to know what constitutes a solid analogy and what information needs to be captured to effectively help the reasoning process. Since domain analysis is labor and knowledge intensive, it will be more cost effective if the result or experience can be applied to other domains as well. The information needed for analogical transfer can be identified as "by-products" during the domain analysis process without much overhead, provided that we know what information needs to be captured and analyzed.

The first objective of this article is to bridge the gap between domain analysis and analogy by presenting an integrated approach to reuse. Domain analysis is labor intensive and time consuming. The effort required for domain analysis could be high. The approach will be more cost effective if the results of domain analysis can be further applied across domains. The paper presents an approach called analogy-based domain analysis (ABDA). Figure 1 illustrates the basic principle of the approach, which is modified and expanded from Moore and Bailin (1991), and Lung and Urban (1993). Moore and Bailin demonstrate two roles of reuse, that is, system development and domain analysis. Domain analysis, which is the "supply side" of reuse, provides information and assets for application systems development, which in turn constitutes the "demand side" of reuse. The emphasis of this article is domain analysis and the other demand side, i.e., analogy analysis, with an aim to support development across domains.

Analogy has been applied to software engineering problems. However, some critical factors advocated in analogy are not considered in software analogy. As a result, analogy mapping is primarily or only based on similarities which have different levels. In addition, two entities that share similarities may have significant differences in other aspects. If we only reason based on the surface similarities for the target without recognizing the higher-level differences between the base and the target, the result could be costly in the later maintenance phase or may be harmful. Software reuse must consider the quality issue. Reusing poor quality software also has the worst impact on productivity (Jones 2000). The rationale is to identify the information needed and build a sound model for software analogy. We carefully examine the requirements, constraints, and models developed in the analogy community to satisfy the objective.

Demand Side (intra-domain)	**Supply Side**	**Demand Side** (inter-domain)
System Development	Domain Analysis	Analogy Identification
Development with Reuse	Analysis for Reuse	Transfer by Reuse

Fig. 1 Roles of reuse (adapted from Lung and Urban 1993)

The second objective of this article is to bring to the attention of the software engineering community some important concepts and methods presented in analogy and knowledge management (KM). There have been considerable research and empirical studies conducted in these two areas. Furthermore, the basic concept of reuse and analogy is similar. Although software is more complicated than the examples demonstrated in analogy, there are lessons and experiences reported in the analogy community that the software engineering community can benefit from. Based on the study in analogy and experiences reported in domain analysis, a set of modeling techniques are presented to meet the needs for software analogy and domain analysis. The paper describes generic modeling techniques instead of focusing on specific methods or notations. However, other modeling techniques that are suitable for the applications or have already been adopted can also be applied to complement the approach or provide different views.

The remainder of this article is organized as follows. Section 2 demonstrates an overview of various aspects discussed in analogy. Section 3 describes extensions of analogical studies for the purpose of developing a model to support software analogy. The ABDA approach and examples are then illustrated in Sect. 4. Section 5 presents a case study using the modeling techniques. The last section consists of a summary and a discussion of future directions.

2 Overview of analogical problem solving

The overview includes the critical concept of analogy. Analogy has relevance to case-based reasoning (CBR). In fact, this term is sometimes used as a synonym to CBR. However, analogy is also often used to characterize methods that solve new target problems based on past cases (base) from a *different domain*, while typical case-based methods emphasize on indexing and matching strategies for single-domain cases (Aamodt and Plaza 1994). Similarly, analogy is also related to KM. Knowledge management promotes the creation, sharing, and learning of knowledge within an organization from the business perspective. CBR and KM are closely related. CBR can be used to assist with KM processes and KM techniques can assist the CBR process.

Analogy is a vital ingredient in scientific discovery. Many scientific theories have been derived by discovering or observing analogies between two problems. Analogy also has been studied in the areas of psychology, cognitive science, artificial intelligence, education, and philosophy (Gentner et al. 2001; Sowa and Majumdar 2003). Comprehensive surveys or collections of research works on analogical reasoning can be found in (Hoffman 1995).

The concept of analogy has been addressed or applied to software engineering (Silverman 1985; Finkelstein 1988; Miriyala and Harandi 1989; Maclean et al. 1991; Neal 1990; Maiden and Sutcliffe 1992, 1993; Harandi 1993; Gennari et al. 1995; Lung and Urban 1995a, b; Spanoudakis and Constantopoulos 1996; Massonet and van Lamsweerde 1997; Chiang and Neubart 1999; Pisan 2000; Idri et al. 2002; Lung et al. 2002; Lung 2002; Grosser et al. 2003; Bjornestad 2003; Yimam-Seid and Kobsa 2003; Hamza and Fayad 2005).

Most of the papers in software engineering emphasize the identification of similarities between the base and the target and draw the conclusion of analogy between the two domains. Those approaches often are based on one or limited analogy models. This type of approach may be fine for general problem solving or knowledge transfer. However, reuse of software needs to be built on a solid foundation or high-quality product. Some factors that are vital in analogy are not thoroughly considered in those papers. For example, differences can be as important as similarities in analogical reasoning. Similarly, both high-level structure and low-level components are important in analogy mapping.

The paper presents a model through various critical concepts reported in the area of analogy. The remainder of this section briefly presents influential analogy theories in cognitive science: structure-mapping, high-level perception, pragmatic theory, and taxonomy of semantic relations. These theories address various constraints guiding the analogical mapping between problems and provide a broad view of analogy in problem solving.

2.1 Structure-mapping

The structure-mapping engine (SME) (Falkenhainer and Forbes 1989; Gentner 1989) is both theoretically and empirically significant and the theory provides some major breakthroughs in the current understanding of analogical reasoning (Kedar-Cabelli 1988). The central idea in structure-mapping theory is the principle of systematicity. The systematicity principle states that analogy is a mapping of "systems of relations governed by higher-order relations with inferential import, rather than isolated predicates" (Gentner 1989). In other words, the principle emphasizes on a relation of relations (higher-order relation) rather than just a relation of objects (low-order relation).

2.2 High-level perception (HLP)

Contrary to the SME approach, the HLP theory (Chalmers et al. 1992; Morrison and Dietrich 1995) views analogy as a bottom up process. In other words, low-level perception will be used to build a representation. Those low-level perceptual processes interact with high-level concepts which help the analogical processing.

2.3 Pragmatic approach

Holyoak and Thagard (1989) have pursued the goal-related concept and proposed a pragmatic approach for analogy. The approach incorporates three types of constraints into the analogical procedure. These constraints are structural consistency, semantic similarity, and pragmatic centrality.

2.4 Taxonomy of semantic relations

Studies in analogy show that analogy may be processed differently based on the types of relations. Hoffman (1995) also points out that analogy is not just the recognition of similarities, but distinctions and differences are also important. Bejar et al. (1991) conducted a survey on the analysis of semantic relations and analogy, and derived a taxonomy to group semantic relations. With the taxonomy, the relations can be decomposed into common aspects that the systems are sharing and aspects that the systems differ. Thereby, relations can be better understood at a higher level and analogical mapping between different domains can be facilitated.

The taxonomy consists of two main types of semantic relations: intensional and pragmatic relations. Intensional relations are based solely on attributes of two objects or items. These types of relations can be understood by interpreting the meaning and characteristics of the objects. Pragmatic relations require knowledge about the problem area the objects involved. The knowledge reflects how the objects are interrelated in the system. Table 2 shows the types, classes, and examples of semantic relations. Each class further consists of the specific relations as members.

A thorough list of specific members for each family relation is presented in Bejar et al. (1991). Not all of the relations may be useful in software modeling. Nevertheless, the list can serve as a guideline for classifying semantic relations. The classification of relations can be used as criteria for understanding and comparing relations in different domains to reveal underlying similarities and differences.

In summary, the structure-mapping theory looks at analogy from a horizontal perspective. On the other hand, the HLP views analogy from the vertical point of view (Morrison and Dietrich 1995). The pragmatic approach and the semantic relations are used as constraints in validating the analogy.

Table 2 Semantic types of relations and examples (adapted from Bejar et al. 1991)

Semantic type	Semantic class	Specific members or examples
Intensional	Class inclusion	Employee: person; shape: circle
Intensional	Similarity	Car: auto; buy: purchase
Intensional	Attribute	ID: person; price: product
Intensional	Contrast	Receive: send; buy: sell
Intensional	Nonattribute	CPU: storage; running state (process): waiting for an event
Pragmatic	Case relations/event	Pilot: aircraft; doctor: patient
Pragmatic	Cause–purpose	Circular wait: deadlock; Gasoline: car
Pragmatic	Space–time	Judge: courthouse; Retirement: pension
Pragmatic	Part–whole	Object: component—car: engine Collection: Member—company: department Mass: Portion—water: drop Event: feature—aircraft: pilot Activity: State—flying:landing Item: Topological Part—room:corner
Pragmatic	Representation	Building: blueprint; red light: stop

3 Development of a software analogy model

Software systems, however, are normally much more complex than the examples presented in the current analogy related literature, though analogy is aiming toward large applications (Forbus 2000). Existing analogical approaches to software reuse also fall short to provide sufficient information to support the reasoning and mapping process. To effectively promote software reuse, current analogical models must be extended for more complex and difficult analogies. In order to achieve this goal, the extended model should provide multiple viewpoints to describe the problem domain more comprehensively. The objectives of this section are to:

1. add additional constraints to an analogical model for software reuse;
2. describe essential aspects for modeling software systems to support software analogy; and
3. develop a process of reusing existing software solutions using analogy.

3.1 Constraints of analogy for software reuse

In order to satisfy some of the requirements, several constraints are used as guidelines and enforcements to lead to quality analogy mappings. In Sect. 2.3, three different types of constraints have been introduced, namely, syntactic, semantic, and pragmatic constraints. In addition to these three constraints, the concept of system dynamics, such as events, triggers, time aspects, object behaviors, and business rules, must be examined as another constraint in an analogical approach for software reuse.

In modeling a system, we need to describe its dynamic features, how it accomplishes tasks, not just its static structure. The reason for dynamic features is due to the increasing complexity and interconnected nature of software. Moreover, in real world applications, user requirements are constantly changing and business rules keep evolving, even if most objects and the high-level relations of the objects stay the same. In addition, personnel shuffling often causes another problem in software evolution. To support software reuse, the "embedded" dynamic information must be identified and validated before reuse occurs. System dynamics information can help better understand problems and prevent using "look alike", but non-analogous solutions for the target.

3.2 Components for analogy-based reuse

The components of reusing software assets are relatively well known in software reuse (Biggerstaff 1992; Krueger 1992). The main phases involved in the process include abstraction, identification, selection, adaptation, and integration.

Based on the empirical observations conducted in analogy and the process of reusing software assets reported in the reuse community, an analogy-based reuse process model is presented in this section, as shown in Fig. 2. The process consists of several components, but it does not have to be carried out in a strictly sequential manner. On the contrary, the process may be highly iterative and the steps may interact with one another in various ways. However, the framework serves as a useful conceptual model for the overall analogical process. Those components share a lot of similarities with that of reusing software assets (Biggerstaff 1992; Krueger 1992). This paper emphasizes on representation, mapping, and generalization/classification.

3.2.1 Representation

Analogical reasoning can be made efficiently and effectively if there exist suitable underlying representations. Rich representation schemes can support the identification of objects and features of the problem domain and thus facilitate the understanding of the domain. Moreover, good representation schemes can support all other phases of analogy processing. Key representations that are necessary for analogy will be discussed in the software analogy modeling in Sect. 3.3.

3.2.2 Mapping

Given critical artifacts, mapping is the process to compare the base and the target and select relevant base properties over the target. Mapping is the core of analogical transfer and must consider the syntactic, semantic, and pragmatic aspects of a domain. This phase has a crucial role in software reuse. Many papers in software analogy consider only the similarities between the base and the target problems. This paper argues that differences are as important as similarities and must be evaluated in the mapping process, especially for software reuse. If the differences are not carefully identified and evaluated, reuse of similarities alone could cause more downstream problems and could exert negative impact on software productivity.

3.2.3 Generalization and classification

Generalization or abstraction has a central role in any reuse approach. With abstractions, software developers can figure out more easily what the asset does, when the asset can be reused, and how the asset can be reused. In order to support software analogy across domains, generalization also deals with the abstraction of common structures and concepts between the related domains.

Classification is one of the most fundamental methods used in science and engineering to help people better understand a particular system. Classification has a great influence on problem identification and retrieval. Numerous classification schemes and clustering techniques have been reported in the literature. Prieto-Diaz

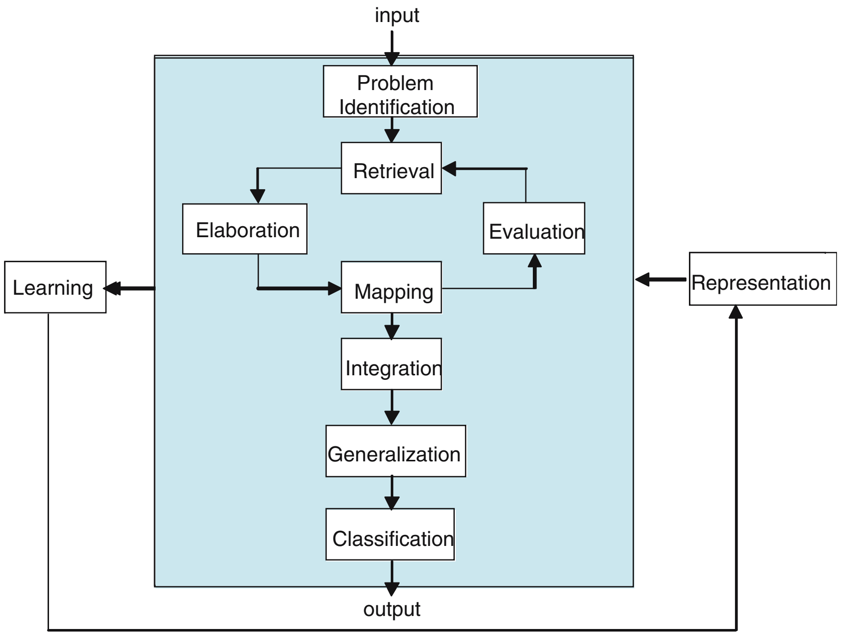

Fig. 2 Components of software analogy

(1991) applied the faceted classification scheme in the library science to classify software components. The approach has been adopted by many researchers in the software engineering and KM communities. A recent example is Vitharana et al. (2003). Section 4.3 presents a layered faceted classification scheme for domain models classification.

3.3 Software analogy models

This section addresses the ‘‘supply side’’ of reuse, i.e., domain analysis, as illustrated in Fig. 1. The models that should be used to describe a system are the subject of this section. Those models or similar models have been extensively discussed in the software community. In fact, many of them have been proposed or used in requirement or domain analysis. The most widely discussed software modeling technique is UML (Unified Modeling Language) (UML 2005) which is composed of a set of models. This article does not attempt to create new models; rather, we address what existing models can be adopted and how to make use of them collectively. In addition, this article puts emphasis on generic concepts of modeling as opposed to specific representations and from the analogy perspective. Selection of a specific technique depends on the application and the environment.

To capture the important aspects of software systems, a model is needed for each of the following items:

1. *Objects*. Object modeling describes the objects in the problem area, their attributes, operations, and relationships with other objects.
2. *Functions*. Functional modeling specifies what the system does, as well as the data transformations and tasks performed by objects.
3. *Relations*. Relational modeling deals with two tasks: (i) classification of semantic relations for the purpose of understanding and comparing lower-order relations between components; (ii) generation of a higher-order causal relation.
4. *System dynamics and goals*. System dynamics modeling describes the information that will potentially change over time. Goals or purposes are constraints for selecting only relevant analogical information.

The idea of object modeling is similar to object-oriented analysis. In object modeling, objects and the relationship between objects are identified, and static structures of the objects are constructed. Functional modeling reveals tasks that the objects and the system perform. Functions are salient characteristics of a system and most systems can be recognized by the functions they realize. Relational modeling emphasizes semantic and causal relations. For complex problems, static structures of objects or functions do not provide sufficient knowledge for analogical reasoning. Semantic relation modeling identifies ways in which the relations are the same and ways in which the relations are different.

Causal relations represent high-order relations between objects. Causal relation modeling supports the application of Gentner’s structure-mapping principle. The relational modeling can help compare relations and evaluate analogies. A dynamic model should be able to represent states, events, object behavior, and rules in an application. Moreover, important goals or purposes are included, because different goals may represent specific context and diverse stakeholders concerns. Stakeholders concerns reflect non-functional requirements as well as functional requirements. Non-functional requirements are important, but are often neglected in the modeling phase. Together the system dynamics and goals serve as useful pragmatic constants for analogy analysis.

All four of these different modeling techniques are required for a sound analogical approach for software reuse. As stated earlier, those techniques have been widely discussed. In fact, three of the four models: object, functional, and dynamic (excluding goals or purposes) have also been adopted in requirements analysis methods like recent UML and most domain analysis approaches.

4 Analogy-based domain analysis (ABDA) approach

Having discussed the features of analogy and requirements from the software perspective, we will demonstrate, in this section, the modeling technique, namely, object, functional, relational, and dynamic. The article is more concerned with the modeling concepts of each technique than specific notations. Different users may adopt different notations for specific environments. This article also puts more emphasis on those aspects which differ from or extend other modeling methods.

The ABDA approach rests upon existing domain analysis concepts, particularly generic/specific modeling technique (Lung et al. 1994) and FODA (Kang et al. 1990), and is expanded to satisfy the needs postulated in the analogy research area. ABDA consists of three main phases: definition and identification, domain modeling, and domain architecture construction. The approach is depicted as follows:

1. Domain definition and identification,
   - Identify domains and define domain boundaries,
   - analyze domain problems and define objectives, and
   - conduct the context analysis.
2. Domain modeling
   - Develop product models, including object, functional, relational, and dynamic models,
   - generalize and classify product models, and
   - Evaluate models with use cases and scenarios, and classification.

3. Domain architecture construction
   - Define process interaction,
   - define reference architecture, and input and output, and
   - refine and evaluate scenarios and the reference architecture.

Figure 3 demonstrates the ABDA process by showing activities, products, product flow, and information flow based on the domain analysis classification and comparison approach proposed by Wartik and Prieto-Diaz (1992). ABDA integrates bottom-up and top-down concepts. Bottom-up analysis supports the identification of objects, operations, relationships, and featural similarities in a class of systems, while top-down synthesis facilitates the construction of knowledge structures and domain models or architectures and promotes high-level reuse instead of just code reuse. Knowledge structures or architectures can be more abstractly represented than objects and operations and can possess more information than features. The high-level abstractions are necessary due to the increasing complexity of software systems and the demand to reduce maintenance costs for system evolution. However, the construction of complex architectures must be supported by an object driven, bottom up process (Hanson 1983). The following sections illustrate the critical factors involved in each step of the modeling process.

4.1 Definition and identification

During the definition and identification phase, purpose, completion criteria, user's needs and scope are defined. These considerations can help the domain analyst recognize what areas of a domain to focus on. The point to recognizing application domains is to identify patterns in existing applications. Patterns may occur at various levels: program code, design, architecture, or business area. Defining domain boundaries can determine the range of the domain to be analyzed and the model to be generated. Features in a domain are also bounded by the domain definition. For a domain interrelated with other

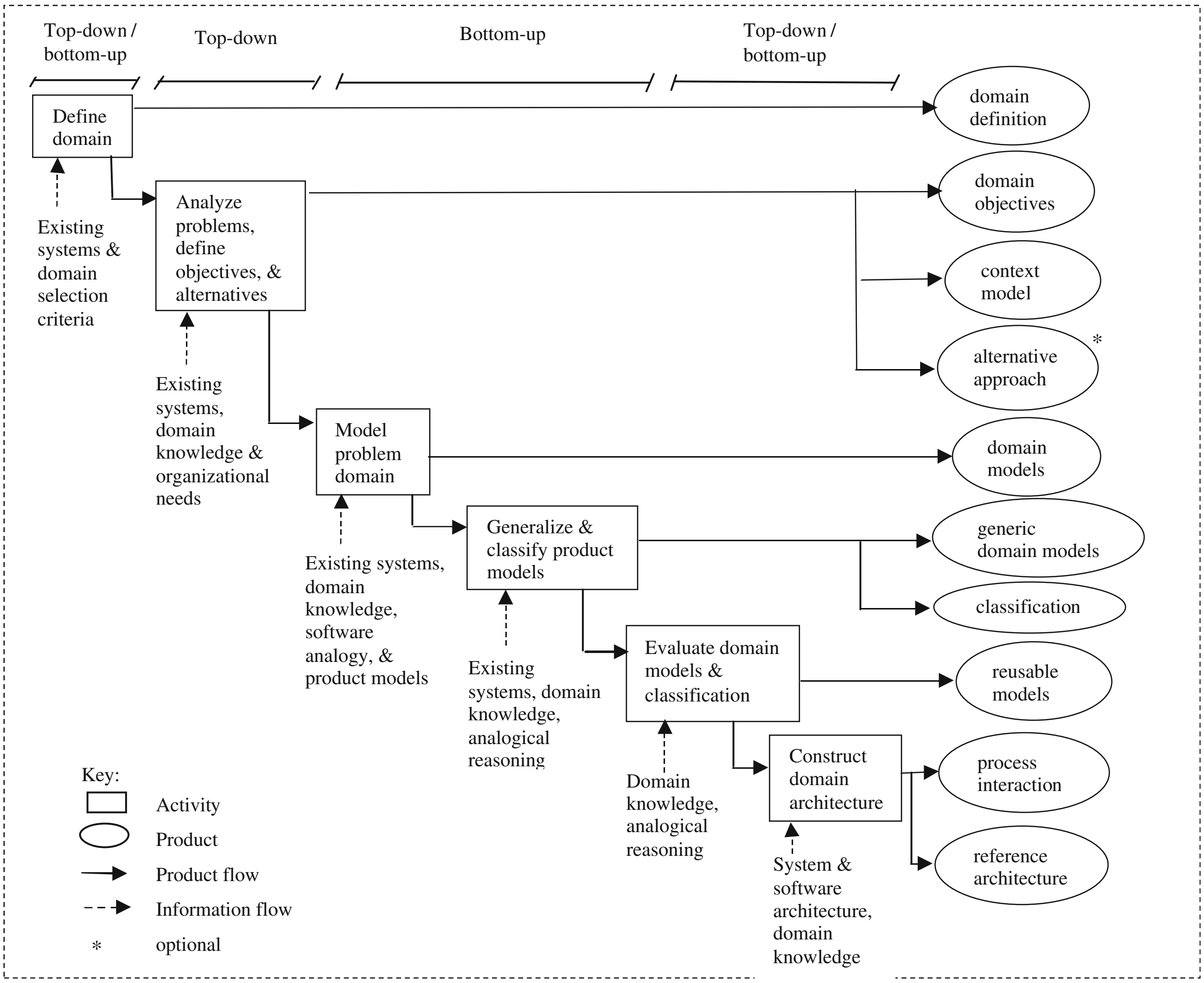
Top-down / bottom-up
Top-down
Bottom-up
Top-down / bottom-up
Define domain
Existing systems & domain selection criteria
Analyze problems, define objectives, & alternatives
Existing systems, domain knowledge & organizational needs
Model problem domain
Existing systems, domain knowledge, software analogy, & product models
Generalize & classify product models
Existing systems, domain knowledge, analogical reasoning
Evaluate domain models & classification
Domain knowledge, analogical reasoning
Construct domain architecture
System & software architecture, domain knowledge
domain definition
domain objectives
context model
alternative approach
*
domain models
generic domain models
classification
reusable models
process interaction
reference architecture
Key:
Activity
Product
Product flow
Information flow
* optional

Fig. 3 Analogy-based domain analysis (ABDA) process

domains, the boundaries are best defined in an operational way (Simos 1991).

Several researchers (Simos 1991; Biggerstaff 1992; Arango et al. 1993) have advocated some key factors to consider in selecting and scoping an application domain. Those factors include narrow breadth, stability, maturity, longevity (the domain should be useful for a long period of time), and economy (the domain should have potential for payback through reuse).

The next step is to analyze domain problems and define objectives. The main purpose of this step is to find out what needs to be accomplished. The user role and system characteristics are essential in identifying the objectives and analyzing a problem. For mature and well-understood domains, this step may be relatively easy. For large, complex, or newer domains, an incremental approach is advocated to gradually acquire knowledge.

Based on the defined scope and identified objectives and stakeholder's needs on the problem domain, a paradigm shift may need to be adopted if the objectives and needs cannot be satisfied by using the conventional method. The concept is optional depending on the specific problem space. Lung et al. (1994) presented an empirical study on simulation domain modeling in manufacturing, which addressed the concept. Traditionally, the modeling of manufacturing systems through simulation is complicated and time-consuming because of the intensive knowledge and labor required. The primary problem with traditional simulation modeling is the lack of reusability of the simulation models, despite the fact that there are similarities among simulation models (Lung et al. 1994). The modeler views each simulation project as unique and constructs a specific model to meet the objectives. As a result, cost overruns and poor performance are usually associated with the simulation building process. In addition, simulation models often need to be written by outside experts instead of manufacturing engineers who know best what they want to simulate.

The new design, based on intensive domain analysis of 150 discrete manufacturing models, was a generative framework for the user to select appropriate generic simulation models from which a specific model can be instantiated. The above article also described the subsequent development of the domain architecture as a result of the new paradigm. The concept has also been adopted more frequently as generative programming techniques become more accepted.

After the domain and its boundaries have been identified and high-level problems and objectives have been recognized, the last step in this phase is to develop a context diagram. The context diagram reveals the interactions between the domain and its external environments, and the information that flows between the domain and the environment. The context diagram is also a channel to introduce objects in the modeling stage.

4.2 Domain modeling

This article concentrates on the domain modeling phase. To model a problem domain, we follow the line of thought postulated in the analogy discipline. This phase consists of four modeling techniques: object, functional, relational, and dynamic. These four modeling techniques are executed iteratively rather than in a strict sequential order. For example, when a new object is uncovered later in the process or if the relation between two objects gets evolved, the previous models generated may need to be revised. Those four modeling techniques are demonstrated in the remainder of this section. Two sets of examples are used for illustration; one includes the car rental problem and the library problem; the other contains problems in both discrete and continuous manufacturing.

4.2.1 Object modeling

In object modeling, main domain components and their relationships are identified. This concept is also the focal point of object-oriented methods and various domain analysis approaches. We are not discussing the object-oriented analysis methods in detail, since they are widely used. The main difference between domain analysis and system analysis or object-oriented analysis is that domain analysis deals with multiple systems instead of a single system. Czarnecki and Eisenecker (2000) compared and contrasted domain analysis and object-oriented analysis in details.

However, there may be significant differences between domains and hence domain analysis and engineering methods. Some methods are more useful for specific domains. ABDA does not impose a particular object modeling method. In fact, multiple methods are often used to capture various domain aspects, e.g., data structures and user interaction. Figure 4 demonstrates an example using the entity-relationship (ER) diagram for object modeling for discrete manufacturing systems. The ER diagram neglects attributes and some detailed parts for simplicity. The ER technique was adopted mainly because the system (the base) was primarily modeled with ER method and yet the method is simple and expressive.

The objects and features identified serve as a baseline for modeling the problem domain. Features can also be classified into different groups: mandatory, optional, and alternative (Kang 1990). Identification of various features is important, because a feature recognized up front in analysis and presented to the designer may help the designer justify and make better decisions. The identification may also yield significant differences in terms of processing or implementation. In cases where features are not easily defined or identified, variability analysis and management among systems is also useful and serves similar purpose.

Figure 5 is an example showing the main components/features in discrete manufacturing. Each component can

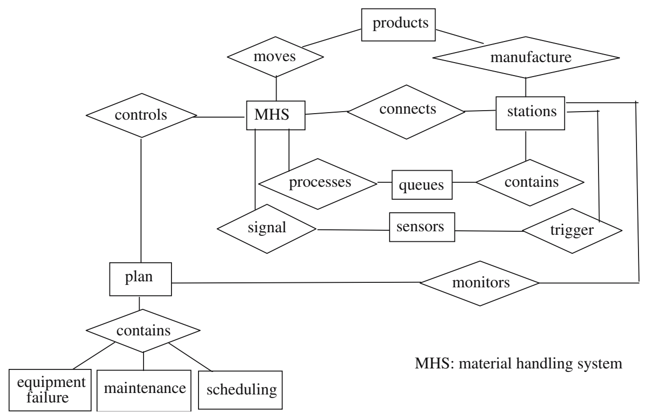

Fig. 4 Entity-relationship (ER) diagram for discrete manufacturing Systems

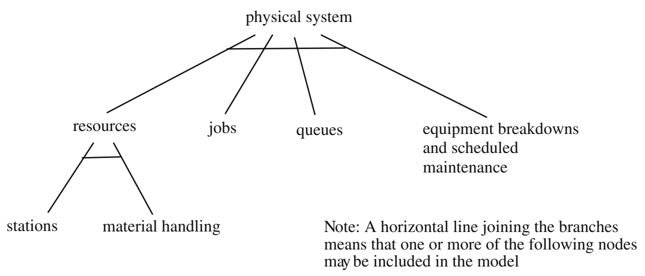
physical system
resources
jobs
queues
equipment breakdowns and scheduled maintenance
stations
material handling
Note: A horizontal line joining the branches means that one or more of the following nodes may be included in the model

Fig. 5 Example showing main features of a manufacturing system (Lung et al. 1994)

be further decomposed into sub-components. Lung et al. (1994) presented a detailed description of modeling various features for the discrete-event simulation in manufacturing problem. The components may look similar to those used in the continuous manufacturing (the target) which is a different domain. This process will not only identify various features, but also facilitate the identification and comparison of the features in another domain. As discussed earlier, differences are also vital in analogical reasoning.

Table 3 presents comparisons among components in discrete and continuous manufacturing. As shown in the table, both domains have similar material handling systems (MHS), whose primary operation is to move and store materials, parts, and products. In a discrete problem, there are usually a group of machine stations, each performing some simple operations such as machining, assembling, or inspecting materials or products. While in the continuous area, the number of machines is fewer, but generally each machine is costly and performs more complex operations. Disassembling and chemical processing are two typical examples in the continuous domain. Because the stations are more sophisticated in the continuous domain, planning for maintenance and equipment failure is more important than that 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 (Lung et al. 2002). The comparison will facilitate the reasoning and mapping process, and the development of an object model for the target problem as shown in Fig. 6.

4.2.2 Functional modeling

Functional modeling deals with the data transformations and tasks performed by objects and systems. Functional modeling captures what a system does and represents the data flow aspects of a system. Functions are related to operations on objects. Thus, the object model developed can be used as a starting point to identify and analyze object operations and system functions. Key system functions are also salient features that can be used to compare different applications or domains. In addition, functional modeling is a well-exploited area in software development and domain analysis.

Table 3 Comparisons of main components between discrete and continuous manufacturing

	Discrete	Continuous
Material handling	Transport and store material	Very similar to discrete domain
Stations	Many, one operation/station, queue before	Few, many operations/station, no queue
Product type	Usually mixed, many could also be 1 product	Product type disassembly (one main product, several sub-products)
Queue	Have queues	Normally no queues
Equipment failure	Same process, minor disturbance, fast recovery, possible alternative	Scarp in process, major disturbance, high reliability required
Planned maintenance	Should do, but often skipped	Very important, hard to stop, more regular
Scheduling	Math difficult, little optimization, predicable, a priori, finite state machines	Sequencing, continuous monitoring (real time), response time, high variance, differential equations
Sensors	Fixed position sensors, flag when an object moves into a range	Normally no sensors

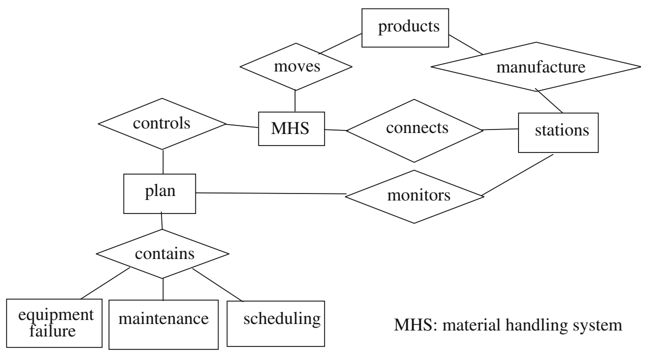

Fig. 6 Entity-relationship diagram for continuous manufacturing systems

Based on the various types of features recognized, commonalities and differences of the applications in a domain are identified. The functional model can be specified using data flow diagrams (DFDs). DFDs show the input, process, and output without detailing how and when the functions are executed. DFDs have been well accepted in the software engineering community for ease of understanding and simplicity. The existing DFDs can be also reused to help analyze a domain.

A high-level DFD for the discrete manufacturing is illustrated in Fig. 7. Based on the artifacts from the base problem and reasoning of the problem domain, a DFD for the target problem is derived, as illustrated in Fig. 8. The seven processes in the target domain shown in Fig. 8 were derived from the base. Processes 1, 4, 5, 6, and 7 are repeated use without modifications. Process 2 is a modification from "monitor the position of all products and identify collisions". Process 3 is changed from monitoring the product position to modifying the product quality. A queue related process (process 8 in Fig. 7) and sensors entity, and all related entities, data stores, and data flows were removed, since normally there are no queues or sensors in the continuous problem.

Each function or process can be even decomposed into sub-processes. These processes in the DFD correspond to operations for objects. In functional modeling, processes related to the alternative or optional features identified in the object modeling phase are carefully examined. Because these features represent differences between applications, separation of these processes can support distinction between possible "black-box" and "white-box" reuse.

For example, the verify_borrower_status process in the library problem can be decomposed into three sub-processes: verify_valid_ID, verify_valid_privileges, and verify_overdue. Verify_borrower_ID can be reused almost without change for any other library system, whereas verify_privileges (e.g., same privileges vs. different privileges for different types of users) and verify_overdue (e.g., one overdue policy vs. incremental policy) are more application dependent, and hence are more likely to be "white-box reused." The analysis and distinction will support downstream quality reuse.

4.2.3 *Relational modeling*

Relational modeling, in this context, is mainly concerned with two tasks: reasoning and classification of low-level semantic relations and identification of higher order causal relations.

4.2.3.1 Classification of semantic relations Bejar (1991) presents the taxonomy of semantic relations. The classification is a useful guideline to start the comparison of "look-similar" relations. Again, the comparison serves as a constraint in order to obtain a more accurate understanding of different or specific situations.

To classify the relations, the emphasis is on the critical objects. Critical objects are those objects identified in the object modeling stage or can be narrowed down to those objects whose removal will cause change in the causal relation. The relations between the critical objects are then analyzed and classified based on the taxonomy. Based on the taxonomy, the following shows the classification of semantic classes followed by specific relations for the discrete manufacturing problem.

Queue: product—Part–whole; Collection: Member
MHS: product—Cause–purpose; Instrument: Intended Action
MHS: queue—Cause–purpose; Instrument: Intended Action
Station: product—Cause–purpose; Instrument: Intended Action
Station: queue—Part–Whole; Object: Component
MHS: station—Cause–purpose; Instrument: Intended Action
Sensor: MHS—Cause–purpose; Enabling Agent: Object
Sensor: station—Cause–purpose; enabling Agent: object
Plan: MHS—Reference; Plan
Plan: station—Reference; Plan

The classification of relations will later be used in the analogical mapping process to compare with that of a target domain. If a different classification occurs in the mapping process, it indicates that careful examination must be taken on the behaviors of objects in the base and target, which will be discussed in the causal modeling and dynamic modeling process. Identical classification of the critical objects may confirm similarity for the target system.

Take a typical manufacturing factory as another example, a machine station and products may have a similar high-order relation as *manufacture* (station, product). However, a lower level relation between the product and its components may be different for different types of factories. In discrete manufacturing, the relation between a part (engine) and a product (car) belongs to the Part–Whole family with the specific feature (Object: Component) (see Table 2), i.e., the second entity is part of the first entity. In continuous manufacturing, a different specific feature is held for Part–Whole relation. Consider a milk factory: 2% milk is one of the by-products of raw milk. The specific secondary relation for this case is (Mass:Portion). The distinction, as outlined in Table 4, indicates that different types of machine stations and control processes are needed for

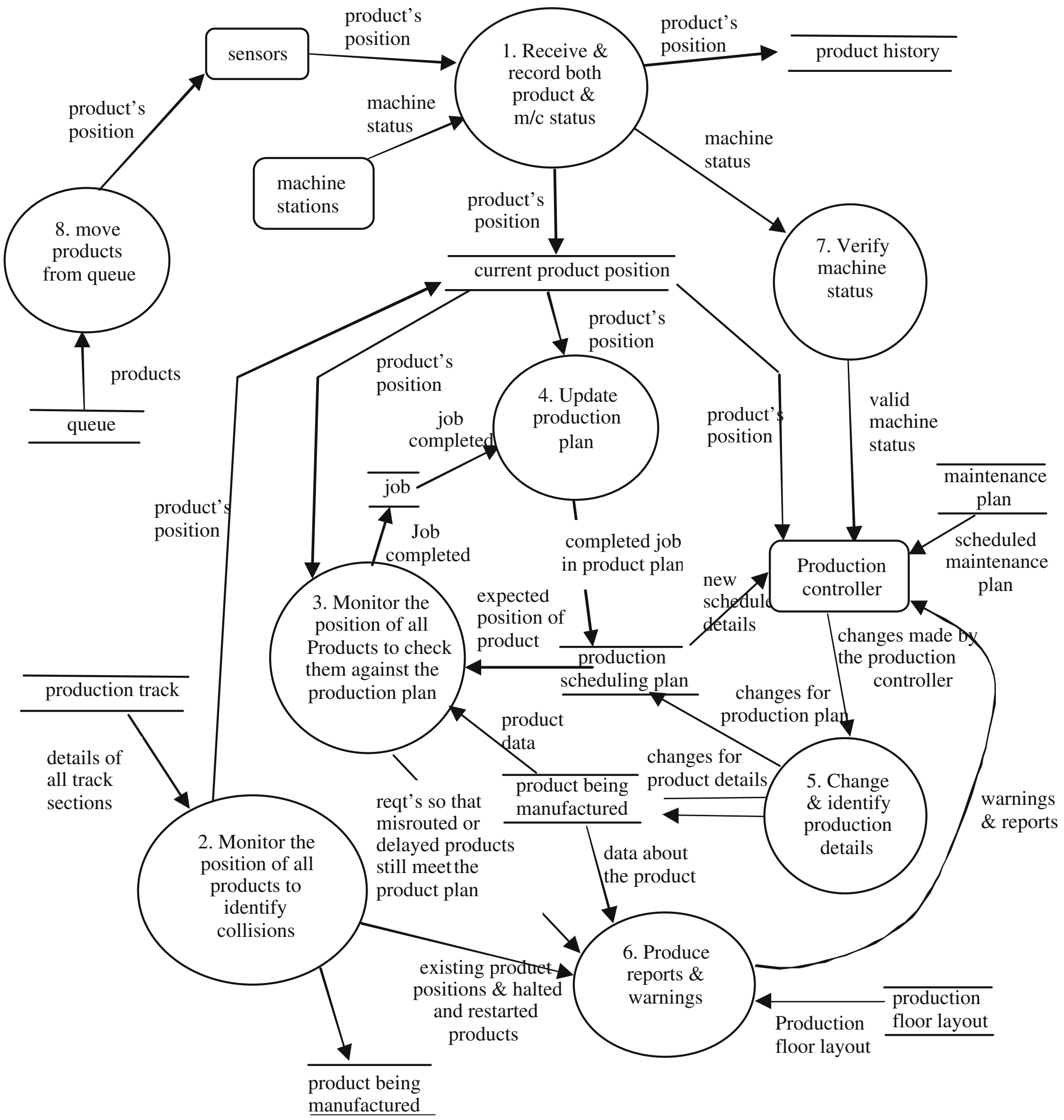
sensors
product's position
1. Receive & record both product & m/c status
product's position
product history
machine status
machine stations
machine status
product's position
8. move products from queue
product's position
current product position
7. Verify machine status
products
queue
product's position
product's position
4. Update production plan
job completed
job
product's position
valid machine status
maintenance plan
product's position
Job completed
completed job in product plan
Production controller
scheduled maintenance plan
new schedule details
3. Monitor the position of all Products to check them against the production plan
expected position of product
production scheduling plan
changes made by the production controller
production track
changes for production plan
product data
changes for product details
details of all track sections
product being manufactured
5. Change & identify production details
warnings & reports
2. Monitor the position of all products to identify collisions
reqt's so that misrouted or delayed products still meet the product plan
data about the product
6. Produce reports & warnings
existing product positions & halted and restarted products
production floor layout
Production floor layout
product being manufactured

Fig. 7 Data flow diagram for discrete manufacturing systems (adapted from Maiden 1991)

these two domains, albeit these two domains share a similar higher order relation.

Even for the same problem domain, a relation between two objects may be changed as a system evolves. For example, the user role may need to be extended from static to dynamic as a new requirement. During the office hours, a user has a business role, whereas the same user may have a personal role away from work as well. The new relation is also explicitly specified and classified as space-time as shown in Table 2. Nevertheless, the object model for the legacy system and the evolving system usually do not represent the dynamic relation. The classification of semantic relations will help compare and discover new requirements for the same problem and explicitly specify the differences.

4.2.3.2 Generation of causal structure of relations The structure-mapping principle, introduced in Sect. 2, postulates that an analogy is a mapping of connected systems of higher-order relations rather than isolated predicates. However, Gentner did not address "how" to draw a network of causal relations. The derivation of a causal model is knowledge-intensive and, unfortunately, there is no explicit guideline to follow.

Roberts et al. (1983) discussed cause-and-effect relationships and presented examples in various problem areas. This research effort adopted some of the notations described by the authors to model causal relations mainly due to its simplicity and genericity. Moreover, the following heuristics were developed for deriving a causal structure of relations:

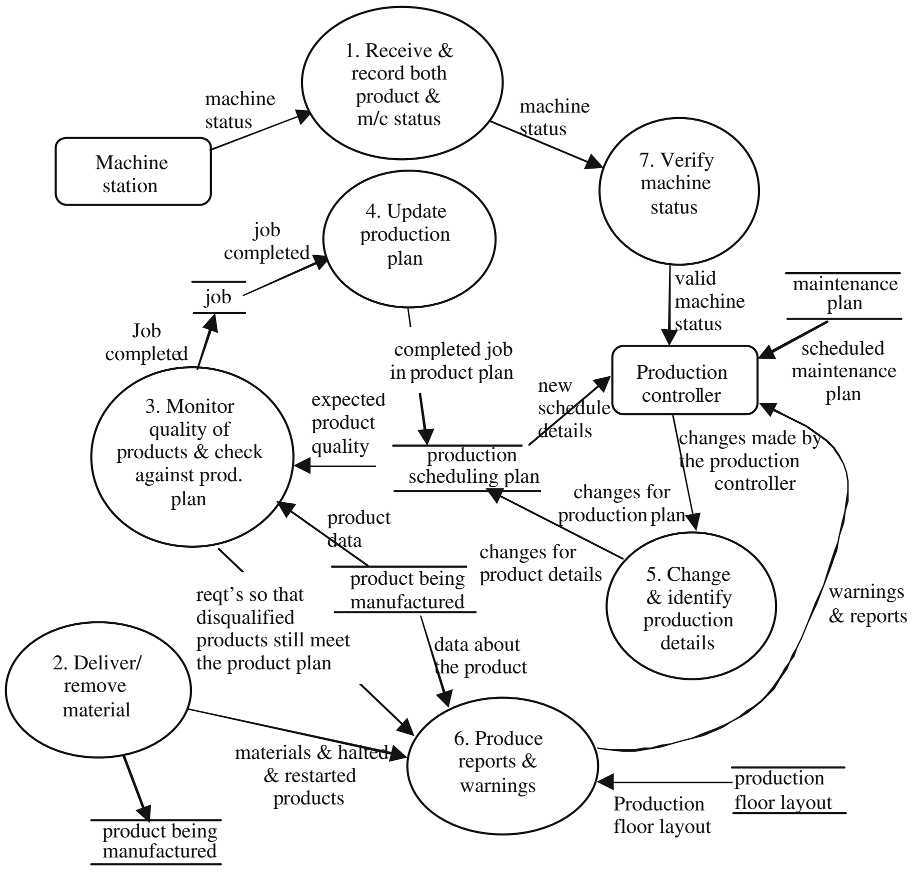
1. Receive & record both product & m/c status
Machine station
machine status
machine status
7. Verify machine status
4. Update production plan
job completed
job
Job completed
valid machine status
maintenance plan
scheduled maintenance plan
Production controller
completed job in product plan
new schedule details
3. Monitor quality of products & check against prod. plan
expected product quality
production scheduling plan
changes made by the production controller
changes for production plan
product data
changes for product details
product being manufactured
5. Change & identify production details
warnings & reports
reqt's so that disqualified products still meet the product plan
data about the product
2. Deliver/ remove material
6. Produce reports & warnings
materials & halted & restarted products
production floor layout
Production floor layout
product being manufactured

Fig. 8 Data flow diagram for continuous manufacturing systems

1. identify key objects and their relationships in the domain (object model);
2. identify primary functions in the domain (functional model); and
3. connect two objects/relationships, or functions that one (object, relationship, or function) is directly influenced or affected by the other (object, relationship, or function).

Step 3 is repeated until all relevant objects, relationships, and functions are examined. A causal structure for the discrete manufacturing problem is demonstrated in Fig. 9. Lower order relations between objects can be explicitly specified, e.g., sensors report product positions. The causal structure represents higher-order relations (relation of relations). The causal structure, which shows the path-centric dependency perspective of the problem area, is complementary to the component-centric aspect illustrated in object model. The causal view describes how the components or tasks are connected. This view can be represented at various levels of abstraction and is useful for communications among diverse stakeholders and for system maintainability, because systems behaviors may evolve as requirements change. However, the static view may stay the same even if system behaviors have been modified.

The causal structure for the discrete manufacturing systems can be reused to identify and transfer to the continuous manufacturing systems with modifications due to the differences between these two domains.

Table 4 Distinction of semantic relations: an example

	Generic semantic class	Specific relation	Example
Discrete manufacturing	Part–whole	Object:component	Car:engine
Continuous manufacturing	Part–whole	Mass:portion	Crude oil:diesel

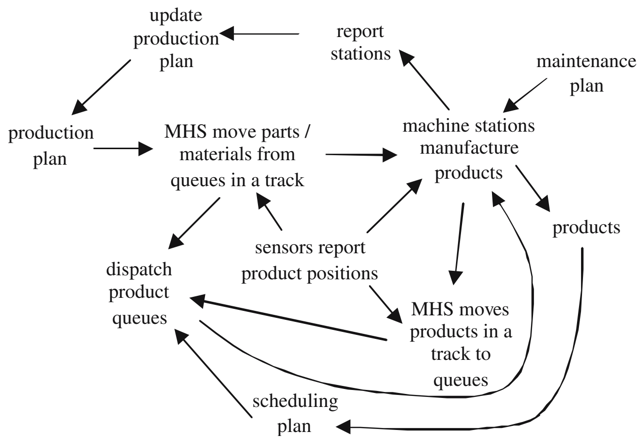
update production plan
report stations
maintenance plan
production plan
MHS move parts / materials from queues in a track
machine stations manufacture products
products
dispatch product queues
sensors report product positions
MHS moves products in a track to queues
scheduling plan

Fig. 9 Causal structure for discrete manufacturing systems

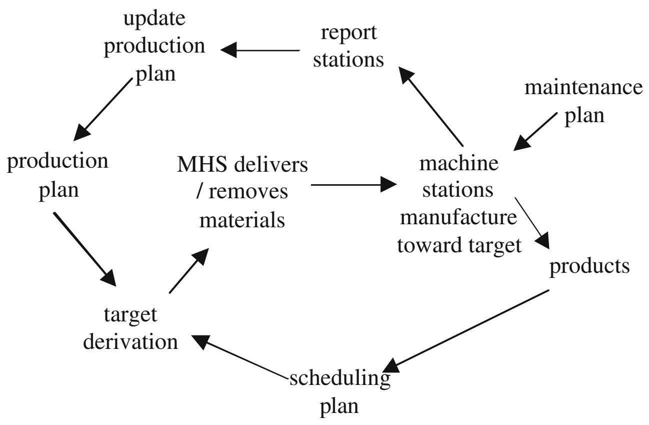

Fig. 10 Causal structure for continuous manufacturing systems

In discrete manufacturing, the time needed to manufacture a certain part of product is predictable and time variation is small. However, in continuous manufacturing, the time needed produce a particular product is dependent on the substance of the raw materials and the target quality. Raw materials coming from different places or the same place but different times may take a slightly different step in order to get the desired products. Time variation in continuous manufacturing may also be high. Based on the causal structure of discrete manufacturing and the differences between the two domains, Fig. 10 presents the causal structure for the continuous manufacturing systems.

The classification of semantic relations of critical objects and the causal structure of high-order relations are primarily used as "pressures" to lead the analogical reasoning process. Higher order causal relations constrain mappings of lower-order relations, which in turn constrain mapping of objects. Semantic relations, on the other hand, can be used to constrain or to make a scrutiny into relation mappings based on different objects.

4.2.4 Dynamic modeling

While causal modeling demonstrates the causes (what), dynamic modeling examines the causes (how). Dynamic modeling, for the "supply-for-reuse" side, rests upon the causal structure of relations developed in the previous step. In fact, causal structure already represents some dynamic behaviors of a problem. Dynamic modeling examines causes that trigger the transition, which includes events, state changes, and conditions. In dynamic modeling, however, the conditions or rules under which a transition is triggered are examined and specified, and the transformations as a result of a transition are also elaborated. Those triggering events can be specified with rules. Rule-based systems are useful to capture and maintain expert's knowledge. Rule-based systems have been extensively studied in the AI and KM disciplines. The simplest form of a rule-based system is a set of IF-THEN rules.

For the discrete manufacturing problem, examples of the external triggering events for the MHS include "material available" and "trigger MHS". The following shows some example rules for the entities MHS and Machine Station.

```
Rules for entity class MHS in discrete manufacturing:
R1:  IF material_available
     THEN enable MHS
R2:  IF trigger_MHS
     THEN check MHS
R3:  check_MHS:
     IF MHS is ready
     THEN enable_MHS
R4:  enable_MHS:
     If from_position and to_position are available and ready
     THEN process_material (from_position, to_position)
```

```
Rules for entity class Machine Station in discrete manufacturing:
R5:  operate_machine:
     IF machine is set up
     THEN check_material
     ELSE set_up_machine
R6:  check_material:
     IF material is available
     THEN load_material
     ELSE wait_until_material_available
     IF loading is done
     THEN start_machine_processing
R7:  start_machine_processing:
     IF machine is ready and material is in position
     THEN process_materail
     IF process_material is done
     THEN unload_material
R8:  unload_material:
     IF product is available
     THEN unload_product
     IF unload_product is done
     THEN trigger MHS
```

For each object or entity class, new rules can be added and existing rules can be modified. The rule-based model can not only reflect the changes in user requirements easily, but also can be used to further distinguish the dynamic aspects of an application domain from similar ones. The rule-based approach can also show different levels of abstractions, which is an important characteristic in analysis and development of software. For example, some operations in the above example can be further expanded.

4.2.4.1 Purpose The purpose or goal is another constraint that is used to examine the causal relations. Purpose is primarily reasoned in the analogical mapping process to choose only relevant causal relations. During the domain analysis process, purposes are also identified in order to support the mapping process. Purposes can be derived from the design objectives outlined in the definition phase and can also be further decomposed to reveal a clearer distinction.

For instance, Maiden (1991) showed analogy between two non-trivial problems: an air traffic controller (ATC) and a flexible manufacturing system (FMS). Both ATC and FMS share common static structures and similar dynamic characteristics (object monitoring). But they have different goals. For ATC, the main goal is to "maintain safe, orderly, and expeditious flow of traffic" (Garot et al. 1987). In FMS, the main concern is how to schedule and coordinate machine stations to gain a high productivity rate and machine utilization, and at the same time reduce throughput time (Talavage and Hannam 1988). The significant difference will lead to different emphasis in reasoning and problem solving.

For our example, the main purposes for the both discrete and continuous manufacturing problems are similar: increase productivity and reduce cost. However, a secondary goal for discrete manufacturing is to avoid collision while moving the parts or products, which is a non-problem in continuous manufacturing. The difference was also illustrated in the data flow diagram.

To map the artifact to the continuous manufacturing, the object model can be used to compare the base with the target. The MHS are similar between these two domains. Hence, the rules can generally be mapped to the target domain. For machine stations, however, modifications need to be made for the target. In continuous manufacturing, loading and unloading parts are not necessary for the stations. Therefore, the rules for machine stations are adapted to suit the new needs, which are depicted below.

Rules for entity class *Machine Station* in continuous manufacturing:

```
R10: operate_machine:
     IF machine is set up
     THEN start_machine_processing
     ELSE set_up_machine
R7:  start_machine_processing:
     IF machine is ready and material is in position
     THEN process_materail
     IF process_material is done
     THEN trigger MHS
```

In addition, purposes or goals can represent non-functional software attributes, which are critical factors for software quality. These attributes can constrain the analogical mapping by selecting appropriate domains or sub-domains for further reasoning. The abstract purposes can be used to derive concrete and explicit scenarios or use cases to validate the domain models.

4.3 Generalize and classify product models

Generalization and classification are vital in analogical reasoning. Generalization is a useful product because a human learns by comparing and generalizing (Mineau 1990). Generalization is also a basis for classification. Meaningful classifications are central to human reasoning and problem solving capabilities (Hanson 1983).

This section presents a layered faceted classification approach for domain models, which is adapted from Lung and Urban (1995a). In other words, the hierarchy consists of multiple layers; each layer in turn consists of a number of facets. The number of layers and facets are expandable and modifiable. Table 5 illustrates the concept of the layered faceted classification with three layers. Layer 1 deals with domain independent facets at the abstraction level. Examples include the application domain, domain abstraction or pattern (higher order analysis patterns or description of primary domain characteristics) (Maiden et al. 1993), domain context architecture, and the key features of the critical objects involved in the problem. The idea of domain abstraction is similar to the problem-class model proposed by Bhansali (1993) and generic data models proposed by Mineau et al. (1993).

Layer 2 encompasses key system functions, object semantic relations, system dynamic, and design goals. Layer 2 is concerned with domain dependent but application independent features. Facets in layer 2 serve to bridge the gap between the generic domain abstractions and specific application domains. These facets are actually abstractions of domain models developed in the modeling phase. The next layer consists of facets that have more detailed information about lower level components and their features. Those features are application dependent.

Each domain analysis pattern can consist of domain specific areas which further consist of multiple application specific facets or even layers. Examples include object allocation, object coordination, and object scheduling. (Maiden and Sutcliffe 1993; Lung and Urban 1995a). Each domain analysis pattern can consist of domain specific areas which further consist of multiple application specific facets or even layers. The concept is demonstrated in Fig. 11 and an example is presented in Fig. 12.

4.4 Evaluation of domain models and classification

The aim of this approach is to support software reuse through analogy for different but analogous domains as well as to promote software reuse in the same application area. Evaluation of the domain models and classification is necessary in order to achieve this goal.

To support reuse, one aspect is to use the artifacts before actual reuse. ABDA and analogy in general assume the existence of the base. In other words, domain models of the base domain are constructed by analyzing existing systems which should have been used before. In one case study on discrete event simulation in manufacturing, a new generic/specific framework was built and actually used in that domain (Lung et al. 1994). Some of the lessons learned from the domain analysis were transferred to the continuous manufacturing problem domain, as illustrated in Sect. 4.2.

The main differences between our approach and other domain analysis methods are the level of abstraction and classification, and the concept of relational modeling. The

Table 5 Layered faceted classification scheme: an illustration

	Facet	Description	Example
Layer 1 (domain independent)	Application domain	The problem application area	Car rental problem
	Domain abstraction or pattern	The high-level analysis patterns or primary domain characteristics/functions	Object allocation and release The pattern supports an analogy for domains that allocate an object to another object (usually agent). The allocated objects are returned after a period of time.
	Analogous domain	Other possible domains that share significant domain similarities	Library systems, reservation systems (hotel, airline)
	Object Type	The resource type of primary objects	Vehicle: reusable, repairable User: customer, agent Staff: agent
Layer 2 (domain dependent, application independent)	Architecture pattern or style	Typical high-level software architecture	Client server model
	Function	What the system does primarily	User can search/reserve vehicles. Staff can search/check out/return/manage/update/verify vehicle fleet
	Relation class	Classes of semantic relations for key objects	User: Vehicle—Agent: Instrument Staff: Vehicle—Agent: Object Staff: User—Agent: Recipient
	Purpose	What the main system goals are. Those goals could be prioritized.	The system provides services to allow users to search and reserve vehicles. The system allows the staff to keep track of the vehicle fleet. The system calculates the rental cost for users
	Trigger	The main cause or event of system dynamics	Manual vehicle reservation Periodical vehicle maintenance
Layer 3 (application dependent)	Rule	Important high-level rules applicable in the application	The user can access the system via the web. The staff has to access the system via designated machines. Only the staff has the authority to update (check out, return, manage) the status of a vehicle
	Non-functional requirement	Special non-functional requirements, e.g., performance	No special performance requirement
	Facility	Underlying important enabling techniques	Web technique, data base management system
	Complexity level	The degree of software complexity or size of the application domain	Small to medium

additional artifacts may not be needed for an application within the same domain, but the information captured is still useful by presenting various viewpoints.

Transferring of knowledge from the base to the target is the first step. The transferred domain models need to be applied to a target system for evaluation before future reuse. The evaluation is conducted by experts in the target domain. In our case study presented in Sect. 4.2, the artifacts for the continuous manufacturing problem were iteratively evaluated by a domain expert.

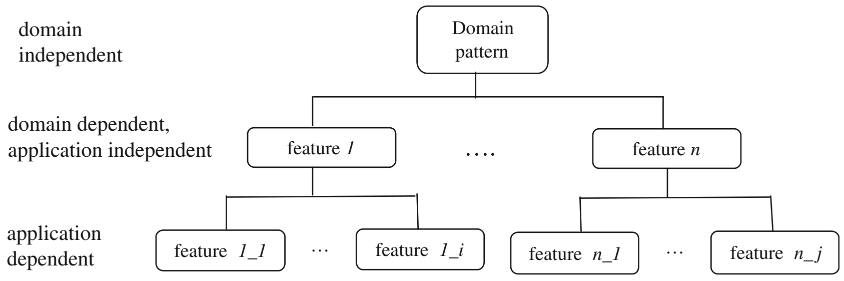

Fig. 11 Layered view of analogous domains and applications

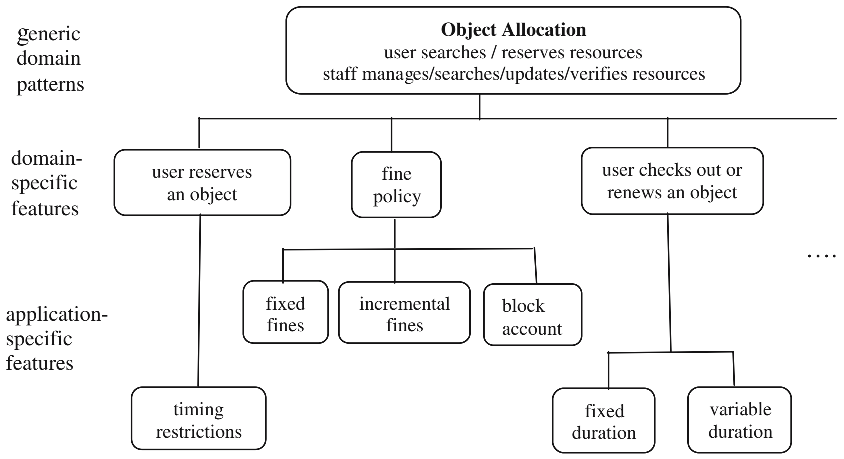

Fig. 12 An illustration of layered classification

4.5 Modeling of domain architecture

Domain modeling focuses on problem space analysis. Based on the analysis, architecture modeling aims at a solution space in the problem domain. This phase is not the primary concern in this research, since this area is closely related to software architecture and tremendous efforts have been put into software architecture. The main difference is that the scope of domain architecture modeling is wider, because multiple systems are considered during the process. Consequently, common structures may need to be identified, which in turn could be used to support construction of a specific architecture.

5 Evaluation and classification of ABDA

This section presents the evaluation and classification of the ABDA approach. Domain analysis is a complex task; hence, evaluation and classification of domain analysis approaches are also complicated. An evaluation was conducted based on criteria in analogy, and domain analysis, since ABDA is closely related to these three areas.

5.1 Evaluation based on requirements modeling

Liskov and Zilles (1975) developed a set of criteria for evaluating specification techniques. This set of criteria is a good starting point for domain analysis approaches, such as ABDA, as a preliminary assessment, since domain analysis centers around requirements engineering. A brief description of the criteria and as assessment of the ABDA approach is as follows:

- *Formality* partially satisfied. A model should be written in a notation which is mathematically sound. ABDA is based on two primary disciplines. One supports design-with-reuse derived from empirically and statistically validated theories in analogy. The other one focuses on design-for-reuse based on software modeling techniques, which may based on formal or informal notations. ABDA does not impose any specific notation.
- *Constructability* partially satisfied. It must be possible to construct models without undue difficulty provided one knows the concepts and techniques involved. ABDA is primarily built upon existing software techniques with an extension of analogy. However, the theme of domain analysis is to study as many applications as possible and derive generic models and architectures. The process and analogical mapping may be timing consuming and knowledge intensive.
- *Comprehensibility* satisfied. A person trained in the technique should be able to study the model and easily understand the concepts captured in the model. ABDA draws as much as possible from areas of existing techniques, such as OO modeling, ERDs, DFDs, or system dynamics, which have been well accepted and widely used.
- *Minimality* satisfied. It should be possible using the model to capture the interesting parts of a concept and nothing more. ABDA is derived to meet the requirements of analogy and software characteristics. Removal of any one of the modeling techniques will make ABDA deficient in satisfying the demands of analogy and software characteristics.
- *Range of applicability* satisfied. ABDA offers the ability to capture both static properties and system dynamics. The features allow ABDA to represent a wide range of applications in general business areas and real-time applications.
- *Extensibility* satisfied. It is desirable that a minimal change in a concept result in a similar change in its model. ABDA is built on object-oriented methods which support separation of concerns and is extensible.

Bordiga et al. (1985) identified some criteria for requirements modeling language. The criteria and the evaluation are highlighted below.

- A good modeling approach should allow the analyst to describe the entities in the domain and changes (events) in the world, and also constraints: satisfied.

- Domain modeling must be able to describe the dynamic aspects of the world and the evolution of the world via time: satisfied. ABDA captures the dynamic aspects of a domain, which can be specified as rules and represented in formal or informal methods. The modeling methods support evolution if domain components, relations, or rules change. Additionally, the generic architecture is a technique that provides adaptability in the abstraction. The feature also supports evolution of an application (Basili 1992).
- The modeling approach should support abstraction: satisfied.
- Domain modeling should include multiple viewpoints: satisfied.
- The modeling approach should be easy to learn, read, and use: satisfied. The criterion is similar to the comprehensibility criterion discussed in the previous assessment. Although ABDA broadens domain analysis concepts and extends analogy approaches for software reuse, it rests upon currently available and commonly used techniques. These techniques are well understood and widely used.

5.2 Evaluation from the analogy perspective

ABDA is mainly derived from the research in analogy. Therefore, it is necessary to evaluate the approach against the criteria advocated in analogy. Three different sets of evaluation criteria were selected and presented as follows.

Sternberg (1977) proposed five criteria for evaluating analogical reasoning methods. Each criterion and a brief evaluation of ABDA are elaborated as follows.

- *Completeness* satisfied. Completeness deals with the coverage of all phases involved in the analogical reasoning process. ABDA extends the process depicted in the paper, which consists of the following phases: encoding, inference, mapping, application, justifications, and preparation and response.
- *Specificity* partially satisfied. An analogy theory is specific if the theory describes in detail each phase involved in the process. ABDA specifically emphasizes representation, elaboration, mapping, generalization and classification. Some steps, such as problem identification and retrieval, may be opportunistic. Usually, a similar analog is either explicitly provided or "somehow discovered" by the analyst. Other steps, like inference and evaluation, integration and modification, and learning, are more subjective and often "embedded" in the process. Learning is a topic that has been discussed intensively in KM. Lessons learned from that area can be applied to ABDA.
- *Generality* partially satisfied. This principle is similar to the criterion of wide range of applicability described above.
- *Parsimony* satisfied. The principle is similar to the minimality criterion presented above.
- *Plausibility* partially satisfied. Plausibility concerns the reasonableness of the approach. ABDA integrates domain analysis and analogy. Domain analysis supports potential software analogy, while analogy facilitates domain analysis activities by providing existing solutions in analogous domains. However, more empirical experiments are needed.

Gentner (1983) presented a group of five dimensions along which measurements can be made on specific analogy mappings. These criteria are discussed as follows with slight modifications to evaluate an analogy approach instead of a specific mapping.

- *Specificity* partially satisfied. The criterion concerns the extent to which the base and the target are understood. ABDA requires that the base be thoroughly understood. The mapping is carried out by comparing and contrasting both domains with various modeling techniques. Generally, the criterion is satisfied. However, if the target is a totally new domain, the degree of specificity for the target may be limited by the specificity of the base.
- *Clarity* satisfied. The criterion deals with the precision of the mapping that the approach can provide. Clarity is violated if the mapping cannot be determined; otherwise, precision of mapping is high. ABDA advocates multiple-level mappings. In addition, the approach addresses both similarities and differences for the mapping phase.
- *Richness* satisfied. Richness refers to, roughly, the potential "quantity" of predicates that can be mapped. However, there is no exact definition for quantity. The main point is that the mapping should encompass multiple viewpoints, which has been emphasized in ABDA.
- *Abstractness* satisfied. ABDA can support mapping of high-order relations as well as low-order relations.
- *Systematicity* satisfied. Systematicity is the mapping of connected knowledge rather than isolated facts. ABDA is an approach of both top-down and bottom-up. The top-down analogical approach proposes mapping of high-order relations which enforces the mapping of low-order relations. The bottom-up approach, on the contrary, constrains high-order relations with lower-order predicates or even attributes.

Silverman (1983) depicted five measures for evaluation of practical analogies. The measures are slightly adapted and are discussed as follows:

- *Procedurality* satisfied. The criterion states that analogy should be formed by following a predefined procedure. ABDA meets the requirement as described in Sect. 3.
- *Contextuality* satisfied. Contextuality is concerned with whether the base is specified in generalized representations rather than in a specific language.
- *Diagnosticity* partially satisfied. The measure refers to the capability to diagnose the analogical mapping

when uncertainties are identified. Silverman suggested the usage of diagnostic parameters which quantitatively indicate a confidence level of the software artifact. In ABDA, diagnosticity is in the mapping phase, i.e., when an uncertainty occurs, the diagnosis is up to the analyst.
- *Temporality* not satisfied. Temporality refers to the evolution of either the schema or the mapping over time. The criterion deals with tracing the history of an analogy as it evolves. ABDA does not address this issue explicitly. However, ABDA supports the evolution of individual domains. This criterion could be tied to the learning phase.
- *Validity* satisfied. If an analogy is drawn based on a well-understood domain and the process, the analogy can be said to a valid one.

The research is primarily concerned with software analogy. Software analogy differs from most other analogy studies in that the potential size may be large and complexity may be high. To address those issues, we consider two more criteria from the software engineering perspective.

- *Group development* not satisfied. Large systems are generally developed by a group of people. Multiple analysts may be working on the same project, but each is responsible for only small portions of the entire project. Problems may arise in several ways. Firstly, different analysts may have different interpretations for the same or similar problem, which in turn may affect the mapping process. ABDA or current analogy approaches provide little or no support for coordination among analysts. Secondly, analysts may not have a clear view of the whole system.
- *Tool support* partially satisfied. Tool support is essential in order to make the knowledge intensive activities efficient and effective. ABDA does not provide any specific tools. However, it is built on existing techniques, tools used in those techniques can also be used. Web and the internet technologies can also facilitate some phases, e.g., problem identification and retrieval, discussed in this approach.

5.3 Classification based on domain analysis criteria

Prieto-Diaz and Arango (1991) illustrated the concept of domain analysis in details and presented a collection of approaches. Czarnecki and Eisenecker (2000) also conducted a survey on various domain analysis approaches and described some updated domain analysis approaches primarily due to the object orientation extension. Readers are referred to these two articles for an overview.

Domain analysis is complicated due to the diverse goals, products, and processes involved. There are many domain analysis approaches; each is targeted to different objectives. Wartik and Prieto-Diaz (1992) presented a list of criteria for classification purpose. The criteria were developed to serve as a common conceptual ground for comparing and contrasting different domain analysis approaches, so that practitioners can determine how to select among these approaches. The section adopts the idea to the classification of domain analysis approaches and presents as follows.

- *Definition of domain* application and business areas. There are two options defined in (Watik and Prieto-Diaz 1992): application area and business area. Application areas views that any set of related programs is a domain, e.g., a stack family or software packages in a specific application, such as numerical array packages and matrix packages (Czarnecki and Eisenecker 2000). Business area focuses on larger systems with the aim of profitability. ABDA can be used in both cases, even if the examples presented in this paper emphasizes more on inter-domain reuse from the business perspective.
- *Determination of problems in the domain* problem and solution-oriented. Problem-oriented approaches first analyze a set of problems and concentrates on problem-level concepts. Solution-oriented methods, on the contrary, examine existing applications to determine common problems. ABDA starts by analyzing and identifying problems, which is followed by identifying and examining analogous features in existing applications and by mapping solutions over to the target.
- *Permanence of domain analysis results* mutable. A process is defined to be permanent if there is no provision for the products to evolve over time; otherwise, a process is mutable. ABDA allows domain analysis products to evolve as new features are identified or new components/systems are developed.
- *Relation to the software development process* meta-process. A process is defined as meta-process if the construction of domain analysis process is one of the objectives. More specifically, domain analysis is part of a meta-process, and the process for domain analysis is separate from the process for application development. ABDA shares similar idea in that it is independent of the software life cycle models.
- *Focus of analysis* decisions and objects/operations. Domain analysis can focus on objects and operations among similar application systems or decisions that the developer needs to make to derive a solution to a problem. In the later case, the analysis centers around not only the similarities, but also the differences among systems.
- *Paradigm of problem space models* both generic requirements and decision model. In ABDA, commonalities are identified and generalized. At the same time, alternatives and differences are also highlighted explicitly to guide the analyst in choosing appropriate software artifacts.
- *Purpose and nature of domain models* process specification. An effective process specifies both the products it requires as inputs and what products it produces as outputs. The process also may describe

how to effectively use those products. ABDA depicts a process which also describes the products it produces.

- *Approach to reuse* systematic. Systematic reuse aims at the future potential of reuse by investing effort up front to build software assets and provide a rigorous process, which is what ABDA advocates.
- *Primary product of domain development* application engineering process. The criterion deals with the most significant product resulting from domain modeling and implementation. The product can be a reuse library or application engineering process. The main product of ABDA deals with the retrieval of analogous domains and presents a process to develop reusable domain models. In addition, the process addresses the mapping and transfer of knowledge to the target domain.

5.4 Practical application experience

We have applied ABDA to two pairs of problems. The first study was to the application of the library problem to the car rental problem. Domain models for the library problem were developed on some existing system models in the literature. The system models were expanded by including variabilities and commonalities, and were generalized to generic models. The domain models were then classified using the layered faceted classification scheme. Finally, the generic domain models were mapped to the car rental problem.

The second study was the mapping from the discrete manufacturing to the continuous manufacturing problem. Domain analysis was conducted in discrete event simulation in manufacturing (Lung et al. 1994), which served as a foundation for the modeling effort for the discrete manufacturing problem. Multiple domain models were then developed and generalized as depicted in Sect. 4. The knowledge was transferred to the new continuous manufacturing area.

In both cases, the development time for the domain models were significantly reduced for the target problems. For the first study, the time spent on the base problem, primarily generalization and classification tasks, was several days. The time spent on the target problem, however, was only several hours. In the second study, the time used for the discrete manufacturing domain modeling was about 2 months. The time that was taken for the target was merely a few days. The base systems for both scenarios were adopted to facilitate the development of a mental or conceptual model for the target. The conceptual model captures the domain knowledge, which significantly improves understanding of the problem area and could evolve into the architectural model, which has the potential to support large-scale reuse (Jacobson et al. 1997).

Some of the concepts were also adopted in a software architecture recovery effort (Lung 2002). The target was a new system developed rapidly for concept demonstration in a then advanced and exploratory network application area. The project had been cancelled, but later there was a need to use the system again. Unfortunately, designers had left the project and there was no documentation. Another similar and well documented system in the traditional telecommunications industry and some relevant design patterns were used as the base. More generalized representations were developed from an existing product and design patterns and were used as a mental model. Two modeling techniques were used for this exercise because there was a high demand on timing and resources. The two primary representations used were similar to the concept of object modeling and dynamic modeling techniques discussed in Sect. 4. During the process, we found that these two domains had much in common. The knowledge gained from the existing solutions was used to help recover the architecture of the target system quickly, which is useful because the target system was exploratory and not well understood.

The proposed approach has extra cost overhead due to additional modeling efforts, primarily generalization and classification. To develop a comprehensive cost model is beyond the scope of this paper. In the following, we present a brief discussion on this issue. Although there is an extra cost, however, the overhead is low provided that domain analysis or something similar (e.g., commonality and variability analysis) has been applied to the problem area, since ABDA is built on top of such an approach.

The following illustrates the cost overhead associated with our approach. Let C_{ra} represent the cost of conventional requirements analysis; C_{da}, cost of domain analysis; and C_{aa}, cost of ABDA. Typically, we have $C_{\mathrm{ra}} < C_{\mathrm{da}} < C_{\mathrm{aa}}$. The overhead of domain analysis and ABDA, on the front end analysis, is $C_{\mathrm{da}}-C_{\mathrm{ra}}$ and $C_{\mathrm{aa}}-C_{\mathrm{da}}$, respectively. It is generally a reasonable assumption that $C_{\mathrm{aa}} - C_{\mathrm{da}} \ll C_{\mathrm{da}} - C_{\mathrm{ra}}$. In other words, domain analysis has much higher cost than requirements analysis. However, once domain analysis is conducted, the extra cost of ABDA is mainly generalization and classification, which should be much lower. This view is from the *development for reuse* perspective.

From the *development with reuse* aspect, a search phase is needed to retrieve existing knowledge. For domain specific areas, the overhead of this phase generally is not high assuming that reusable domain models have been generated. For inter-domain reuse, the cost is higher, since the retrieval phase needs more time and the mapping phase involves further reasoning effort. In addition, analogy is opportunistic from the *development with reuse* perspective even though the method *for reuse*, e.g., ABDA, is planned. In this case, estimation of the cost or return on investment becomes complicated due to various influential factors, such as the complexity, size, and probability of inter-domain reuse.

There are research efforts on the tool support for retrieving analogous domains or components that could mitigate this problem (Bjornestad 2003; Yimam-Seid and Kobsa 2003; Hamza and Fayad 2005). On the other hand, in practice, many problems share similarities (Batory 1994) and same patterns, especially domain-neutral patterns (Hamza and Fayad 2005), exist in many different domains, the likelihood for inter-domain reuse may not be very low. The potential for higher return on investment is to support reuse for novices and new areas. Novices are more likely to reuse knowledge or artifacts from various sources (Desouza et al. 2006). Overall, for novices or for solving new domains where the risk or unknown information is high, a critical need is a sense of the whole rather than the parts. We can gain such a sense by examining similar or analogous problems.

The cost overhead could be lowered if an effort similar to patterns (analysis patterns or design patterns) could be adopted. Patterns could be seen as a scaled down version of domain analysis conducted collaboratively by the community. Knowledge in patterns is reviewed by peers and organized in an open and easy to access environment. The community serves as a virtual organization and provides a continuous learning mechanism. Various ways for discussions, e.g., conferences and discussion groups across organizations, have been formed. Many concrete examples have been captured and documented in patterns, which is also a key factor to analogy. The lessons learned from the patterns community could be adopted for reuse in large to capture more concrete knowledge and artifacts for further reuse.

6 Summary

This article presented an approach, called ABDA, which is an integration of two streams of research areas, namely domain analysis and analogy. The methodology is based on models and empirical evidences reported in analogy and experiences presented in domain analysis and software reuse. Domain analysis and analogy are regarded as complementary tasks in this approach. Domain analysis performed in one problem area supplies quality information for potential analogical transfer to a new or not well-understood domain in addition to reuse within the same domain. On the other hand, analogy studies identify the needs in order to facilitate domain analysis by providing effective information for knowledge transfer. Furthermore, studies and lessons reported in analogy are beneficial to the software community. Examples reveal a productivity of several folds in the analysis phase for our studies by reusing domain models across problem areas.

Analogy does not guarantee a solution if no existing solutions are adequate. Nevertheless, this experience is still useful to identify problems, especially early in the life cycle. Moreover, the previous design process, not just design products, may support the development of the new target problem. The information and experience of failing to transfer knowledge from one domain to a new problem area are also valuable.

There are commonalities in generic solutions across domains (Batory 1994). Two related questions that are vital in analogical transfer are identification of an analogous problem and the classification of application domains. Most papers, including this paper, do not explicitly explain how the base problem is identified. This phase is still in the state of art and is highly dependent on experience and how the artifact is represented. However, much progress has been made lately thanks to the advanced information technology and KM (Bjornestad 2003; Yimam-Seid and Kobsa 2003; Hamza and Fayad 2005). Problem identification and retrieval can be further improved if quality domain models can be effectively classified.

Classification has long been a topic of interest in various areas. Problems that are distinct in syntax may have similar solutions. The problems grouped in one class may be retrieved for a target problem at the same time, where each selected problem may contribute some insights for the understanding and development of the target domain. The challenge is to recognize and group those domains that share significant aspects. In this paper, we demonstrated the layered faceted classification scheme to classify domains and applications.

We presented a primary case study involving two problems. Additional large scale experiments should be conducted. On the other hand, a large problem domain may consist of several "unit domains", where each "unit domain" is a mature area and solutions to the area are well accepted. To tackle a problem domain, we could first decompose the domain and identify analogous "unit domains". The existing solutions and knowledge of the "unit domains" will then facilitate the understanding and construction of solutions for the new domain. With the increasing popularity of patterns, this may be realistic in the future. In other words, documentation of patterns can be enhanced by incorporating more information related to analogous problems.

The modeling techniques demonstrated in this article can also facilitate software architecture construction. Some techniques used in the paper share commonalities with UML, which is widely used to model software architectures and high-level design. Further, the taxonomy of semantic relations may support the identification or distinction of patterns (e.g., architectural patterns, design patterns) by adding more information to the context described in patterns. The methodology should be experimented on different application areas to populate the repository of analogous domains and validate the classification. The taxonomy of relations could also be refined to meet the features of software and OO methods (Opdahl et al. 2003).

References

Aamodt A, Plaza E (1994) Case-based reasoning: foundational issues, methodological variations, and system approaches. Artif Intell Commun 7(1):39–59

Arango G, Schoen E, Pettengill R (1993) A process for consolidating and reusing design knowledge. In: proceedings of the 15th international conference on software engineering, pp 231–242

Basili VR, Caldiera G, Cantone G (1992) A reference architecture for the component factory. ACM Trans Softw Eng Meth 1(1):53–80

Batory D et al (1994) The GenVoca model of software system generators. IEEE Software, pp 89–94

Bejar II, Chaffin R, Embretson S (1991) Cognitive and psychometric analysis of analogical problem solving. Springer, Berlin Heidelberg New York

Bhansali S (1993) Architecture-driven reuse of code in KASE. In: proceedings of the 5th conference on software engineering and knowledge engineering, pp 483–490

Biggerstaff TJ (1992) An assessment and analysis of software reuse. Adv Comput 34:1–57

Borgida A, Greenspan S, Mylopoulos J (1985) Knowledge representation as the basis for requirements specifications. IEEE Comput Mag, pp 82–91

Bjornestad S (2003) Analogical reasoning for reuse of object-oriented specifications. In: proceedings of the 5th international conference on case-based reasoning, pp 54–60

Chalmers DJ, French RM, Hofstadter DR (1992) High-level perception, representation, and analogy: a critique of artificial intelligence methodology. J Exp Theor Artif Intell 4:185–211

Chiang C-C, Neubart D (1999) Constructing reusable specifications through analogy. In: proceedings of symposium on applied computing, pp 586–592

Czarnecki K, Eisenecker UW (2000) Generative programming, methods, tools, and applications. Addison-Wesley, Reading, MA, USA

Desouza KC, Awazu Y, Tiwana A (2006) Four dynamics for bringing use back into software reuse. Commun ACM 49(1):96–100

Falkenhainer B, Forbus KD, Gentner D (1989) The structure-mapping engine: algorithm and examples. Artif Intell 41:1–63

Finkelstein A (1988) Re-use of formatted requirements specifications. Softw Eng J 186–197

Forbus KD (2000) Exploring analogy in the large. In: Gentner D, Holyoak K, Kokinov B (eds) Analogy: perspectives from cognitive science, MIT, Cambridge

Garot JM, Weathers D, Hawker T (1987) Evaluating proposed architectures for the FAA's advanced automation system. Computer 20(2):33–46

Gennari John H, Altman Russ B, Musen Mark A (1995) Reuse with PROTEGE-II: from elevators to ribosomes. In: proceedings of the symposium on software reusability, pp 72–80

Gentner D (1983) Structure-mapping: a theoretical framework for analogy. Cogn Sci 7(2):155–170

Gentner D (1989) Mechanisms of analogical learning. In: Vosniadou S, Ortony A (eds) Similarity and analogical reasoning. Cambridge University Press, Cambridge

Gentner D, Holyoak KJ, Kokinov BN (eds) (2001) The analogical mind: perspectives from cognitive science. MIT, Cambridge

Grosser D et al (2003) Analogy-based software quality prediction. In: proceedings of the 7th workshop on quantitative approach in object-oriented software engineering

Hamza HS, Fayad ME (2005) Stable atomic knowledge pattern (SAK)—enabling inter-domain knowledge reuse. In: proceedings of the 17th international conference on software engineering and knowledge engineering, pp 127–132

Hanson SJ (1983) Conceptual clustering and categorization. In: Kodratoff Y, Michalski R (eds) Machine learning: an artificial intelligence approach, vol. 3. Morgan Kaufmann Publishers Inc.,USA, pp 235–268

Harandi MT (1993) The role of analogy in software reuse. In: proceedings of the symposium on applied computing, pp 40–47

Hoffman, Robert R (1995) Monster analogies. AI Mag 16(3):11–35

Holyoak KJ, Thagard P (1989) Analogical mapping by constraint satisfaction. Cogn Sci 13:295–355

Idri A et al (2002) Estimating software project effort by analogy based on linguistic values. In: proceeding of the 8th symposium on software metrics

Jacobson I, Griss M, Jonsson P (1997) Software reuse architecture, process, and organization for business success. Addison-Wesley, Reading, MA, USA

Jones C (2000) Software assessments, benchmarks, and best practices. Addison-Wesley, Reading, MA, USA

Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21

Kedar-Cabelli S (1988) Analogy—from a unified perspective. In: Helman DH (ed) Analogical reasoning. Kluwer, New York pp 65–104

Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183

Liskov BH, Zilles SN (1975) Specification techniques for data abstractions. IEEE Trans Softw Eng 1(SE-1):7–19

Lung C-H (2002) Agile software architecture recovery through existing solutions and design patterns. In: proceedings of the 6th IASTED international conference on software engineering and applications, pp 539–545

Lung C-H, Urban JE (1993) Integration of domain analysis and analogical approach for software reuse. In: proceedings of the symposium on applied computing, pp 48–53

Lung C-H, Urban JE (1995a) An approach to the classification of domain models in support of analogical reuse. In: proceedings of the symposium on software reusability, pp 169–178

Lung C-H, Urban JE (1995b) An expanded view of domain modeling for software analogy. In: proceedings of the international computer software and applications conference, pp 77–82

Lung C-H, Cochran JK, Mackulak GT, Urban JE, (1994) Computer simulation software reuse by generic/specific domain modeling approach. Int J Softw Eng Knowl Eng 4(3):81–102

Lung C-H, Mackulak GT, Urban JE (2002) Software reuse and knowledge transfer through analogy and design patterns. In: proceedings of the international conference on software engineering research and practice, pp 618–624

MacLean A et al (1991) Reaching through analogy: a design rationale perspective on roles of analogy. In: proceedings of the conference on human factors in computing systems, pp 167–172

Maiden NAM (1991) Analogy as a paradigm for specification reuse. Softw Eng J 6(1):3–15

Maiden NAM, Sutcliffe AG (1992) Exploiting reusable specifications through analogy. Commun ACM, 35(4):55–64

Maiden NAM, Sutcliffe AG (1993) Requirements engineering by example: an empirical study. In: proceedings of the international symposium on requirements engineering, pp 104–111

Massonet P, van Lamsweerde A (1997) Analogical reuse of requirements frameworks. In: proceedings of the 3rd international symposium on requirements engineering, pp 26–37

Mineau GW (1990) Browsing through knowledge: learning by comparing generalization. In: proceedings of the international conference on advanced research on computers in education, pp 261–266

Mineau GW, Godin R, Missaoui R (1993) Induction of generic data models by conceptual clustering.In: proceedings of the 5th international conference on software engineering and knowledge engineering, pp 554–564

Miriyala K, Harandi MT (1989) Analogical approach to specification derivation. In: proceedings of the 5th international workshop on software specification and design, pp 203–210

Moore JM, Bailin SC (1991) Domain analysis: framework for reuse. In: Tutorial on domain analysis and software systems modelling. IEEE Computer Society Press, pp 179–204

Morrison CT, Dietrich E (1995) Structure-mapping versus high-level perception: the mistaken fight over the explanations of analogy. In: proceedings of the 17th annual conference of the cognitive science society, pp 678–682

Neal L (1990) Support for software design, development, and reuse through an example-based environment. In: proceedings of the 5th conference on knowledge-based software assistant, pp 176–182

Neighbors JM (1992) The evolution from software components to domain analysis. Int J Softw Eng Knowl Eng 2(3):325–354

Opdahl AL, Henderson-Sellers B, Barbier F (2003) Ontological analysis of whole–part relationships in OO-models. Inf Softw Technol 387–399

Pisan Y (2000) Extending requirement specifications using analogy. In: proceedings of the international conference on software engineering, pp 69–75

Prieto-Diaz R (1991) Implementing faceted classification for software reuse. Commun ACM 34(5):89–97

Prieto-Diaz R (1993) Status report: software reusability. IEEE Software 61–66

Prieto-Diaz R, Arango G (1991) Introduction and overview: domanin analysis concepts and research directions. In: Tutorial on domain analysis and software systems modeling. IEEE Computer Society Press, pp 9–32

Roberts N, Andersen DF, Deal RM, Garet MS, Shafeer WA (1983) Introduction to computer simulation: the system dynamics approach. Addison-Wesley, Reading, MA, USA

Silverman BG (1983) A good analogy and how to measure it. Technical report, Institute for Artificial Intelligence. The George Washington University

Silverman BG (1985) Software cost and productivity improvements: an analogical view. Comput 18(5):86–96

Simos MA (1991) The growing of an organon: a hybrid knowledge-based technology and methodology for software reuse. In: Tutorial on domain analysis and software systems modelling, IEEE Computer Society Press, pp 204–221

Sowa JF, Majumdar AK (2003) Analogical reasoning. In: proceedings of international conference on conceptual structures

Spanoudakis G Constantopoulos P (1996) Analogical reuse of rquirements specifications: a computational model. Appl Artif Intell Int J 10(4):281–306

Sternberg RJ (1977) Intelligence, information processing, and analogical reasoning: the Componential Analysis of Human Abilities. Lawrence Erlbaum Associates, Hillsdale

Talavage J, Hannam RG (1988) Flexible manufacturing systems in practice: application design, and dimulation. Marcel Dekker, New York

UML (2005) UML resource page, http://www.uml.org/, last accessed date: Oct 17, 2005

Vitharana P, Zahemi F, Jain H (2003) Design, retrieval, and assembly in component-based software development. Commun ACM 46(11):97–102

Wartik S, Prieto-Diaz R (1992) Criteria for comparing reuse-oriented domain analysis approaches. Int J Softw Eng Knowl Eng 2(3):403–432

Yimam-Seid D, Kobsa A (2003) Expert finding systems for organizations: problem and domain analysis and the DEMOIR approach. J Organ Comput Electron Commer 13(1):1–24