Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1999), *13*, 37–61. Printed in the USA.
Copyright © 1999 Cambridge University Press 0890-0604/99 $12.50

A generic ontology-based approach for requirement analysis and its application in network management software

CHAU-YOUNG IVAN LIN AND CHENG-SEEN HO

Department of Electronic Engineering, National Taiwan University of Science and Technology, 43 Keelung Road, Section 4, Taipei, Taiwan

(RECEIVED April 6, 1998; ACCEPTED November 2, 1998)

Abstract

This paper describes a generic ontology-based approach that eases the requirement analysis (RA) work. The approach enables the user to derive method-specific RA tools for different applications. The derivation process is based on a unified framework that contains a software methodology ontology and a knowledge acquisition ontology. The former contains a library of software RA methods and a set of modeling support entities, which use the library to construct methodological knowledge. The latter contains a library of knowledge acquisition techniques and a set of acquisition support entities, which work on the library, guided by the generated methodological knowledge, to extract domain knowledge. The generated methodological knowledge, coupled with the domain knowledge, forms the RA tools. We have demonstrated the use of the approach by deriving an RA tool to assist the system analyst to acquire and formalize a requirement specification for a network management system. This approach facilitates the integrating, sharing, and reuse of software methodologies and knowledge acquisition techniques and alleviates the problems associated with the correct generation of requirement specification for different domains.

Keywords: Methodology Ontology; Acquisition Ontology; Requirement Analysis; Knowledge Acquisition; Knowledge-based Systems

1. INTRODUCTION

Software development is still more artistic than engineering nowadays. While software as art has a certain aesthetic and emotional appeal, the fact is that an artistic endeavor has never been a reliable process, resulting in poor software quality. In a recent study by the Standish Group (Standish Group, 1994), the software industry has incurred several crises, including over 30% of the software projects are canceled before completing, and the average project runs 189% over budget as well as 222% behind schedule. Two major causes are: the inability to work more effectively with the users to better understand their requirements as well as the problems, and the weak engineering discipline in managing requirements. These in turn cause too many requirement errors and frequently changing requirements. Some researches demonstrate that errors in requirements are the most pervasive and most expensive to fix (Davis, 1993; Dion, 1993). Successfully reducing these errors in the early development stage can substantially lower the total number of errors, cut the project costs, and shorten the schedules. There is strong evidence that effective requirement analysis (RA) leads to overall project cost savings. Three primary reasons for this are:

1. Requirement errors typically cost well over 10 more times to repair than other errors.
2. Requirement errors typically comprise over 40% of all errors in a software project.
3. Small reductions in the number of requirement errors pay big dividends in reworking costs and schedule delays.

Although the critical key to the success of a software system is that each stage of a software development process is

Reprint requests to: Chau-Young Ivan Lin, Department of Electronic Engineering, National Taiwan University of Science and Technology, 43 Keelung Road, Section 4, Taipei, Taiwan. E-mail: CYLIN@MS.CHTTL.COM.TW

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

correctly carried out, RA is one of the first and most important steps to affect the development success. Most standards bodies share this viewpoint (ISO 9000-3, 1991; Paulk, 1993). RA should include, but not be limited to, the following: functionality, performance, safety, reliability, security, and privacy. Basically, RA can be viewed as a *modeling process* that deals with what to model and how to do the modeling. Only when the customer needs and relevant constraints for an application are correctly modeled at this stage, can subsequent stages be worked out effectively and efficiently.

To facilitate the work of RA, many software systems have been developed by exploiting proper artificial intelligence (AI) techniques in the development process since the 1980s. Many of them advocate the concept of specific-purpose RA tools. These tools work as an assistant to the system analyst by providing a predefined, analytic method for creating a system specification. Examples include KBSA (Green et al., 1986), containing KBRA (Czuchry, 1988), for automatic programming and algorithm performance optimization; ARIES (Johnson & Feather, 1991; Johnson et al., 1993), supported by KBSA (Johnson et al., 1992), for aircraft control; RA (Reubenstein & Waters, 1991) for knowledge representation and reasoning; and KB/RMS (Binder & Tsai, 1992) for structure analysis. Although these tools have made some progress in RA, problems still remain. First, many of them are too domain specific (or domain bound) and are thus difficult to apply (even after revision) in the domains with different characteristics. Developers usually must redesign or refine them before applying them to other domains. Moreover, some enterprise-wide systems involve multiple types of domains (e.g., data transaction, real-time control, flow control, knowledge reasoning, etc.). They cannot properly be supported by a tool with only a single analytic method. This implies that a multistrategy approach that integrates various analytic methods to support the RA of a large-scale heterogeneous software system is strongly needed. Unfortunately, the granularity of integration in current RA tools seems too big to support flexible and intricate integration. In a word, current tools are fairly poor in method sharing and reuse.

To properly attack these problems, we resort to the concept of *ontology*. The word "ontology" has a long history in philosophy, where it refers to a systematic account of existence. This account does not depend on any particular language. In the AI community, researchers have given definitions with slight different emphases to ontology (Alberts, 1993; Wielinga & Schreiber, 1993; Gruber, 1995; Schreiber et al., 1995; van Heijst et al., 1997; Guarino, 1997, 1998). One of the most cited definition is "an ontology is an explicit specification of a conceptualization" (Gruber, 1995). Other important features mentioned in these definitions include "shared conceptualizations," "intensional structure," and "logical theory." From these definitions we conjecture that *an ontology is a shared logical specification that defines the ontological commitment to a particular conceptualization*. Specifically, an ontology refers to a shared engineering artifact consisting of a specific vocabulary that describes a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary words. An ontology level of knowledge representation thus constrains the semantics of the concepts and characterizes the meaning of the basic categories used to describe the reality (domain) (Guarino, 1995). Other works that help us to apply ontological engineering in our domain include the research that applied the technique in modeling problems (Lenat & Guha, 1990; Schreiber & Wielinga, 1993; Schreiber et al., 1994) and in supporting problem-solving strategies (Chandrasekaran et al., 1998), which advocates its use in the domains that are beyond object domain knowledge.

Using the ontology concept in the generation of RA tools brings GAMP into being, GAMP for Generic requirement Analysis Meta Platform (Lin & Ho, 1996). Basically, we develop a software methodology ontology and a knowledge acquisition ontology to support GAMP. These ontologies facilitate the *integrating*, *reusability*, and *sharing* of software development methodology and knowledge acquisition techniques, which enables GAMP to generate high-quality RA tools for either domain-specific or enterprise-wide applications.

Figure 1 shows the basic architecture of GAMP. It structures the generation process of RA tools in three models, namely, *method model*, *expertise model*, and *implementation model*. Basically the method model in Figure 1 directs the construction of a particular RA methodology applicable to a given domain with the help of the software-methodology ontology. The expertise model then helps in the acquisition of the knowledge to be used in the RA process for the given domain under the guidance of the RA methodological knowledge just constructed, with the help of the knowledge-acquisition ontology. The implementation model finally deals with the computational requirements of the RA process. By properly materializing these models, we thus generate a domain-specific tool for the RA process, or an RA tool.

The major component of the method model is the *software-methodology ontology* (methodology ontology for short), which contains a *software-method library* (method library for short) and *software-methodology-modeling-support entities* (modeling support entities for short). The method library stores various fine-grained RA-related methods and functions. It represents the experience (methods) of RA of the software-engineering community, which can help the software engineers to construct methodological knowledge for a given application domain. The modeling-support entities guide the software engineers to use the method library to construct domain-specific RA methodological knowledge. This design relieves potential redesign burdens of the software engineers by reusing the fine-grained RA methods previously used in different domains or enterprise-wide systems.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

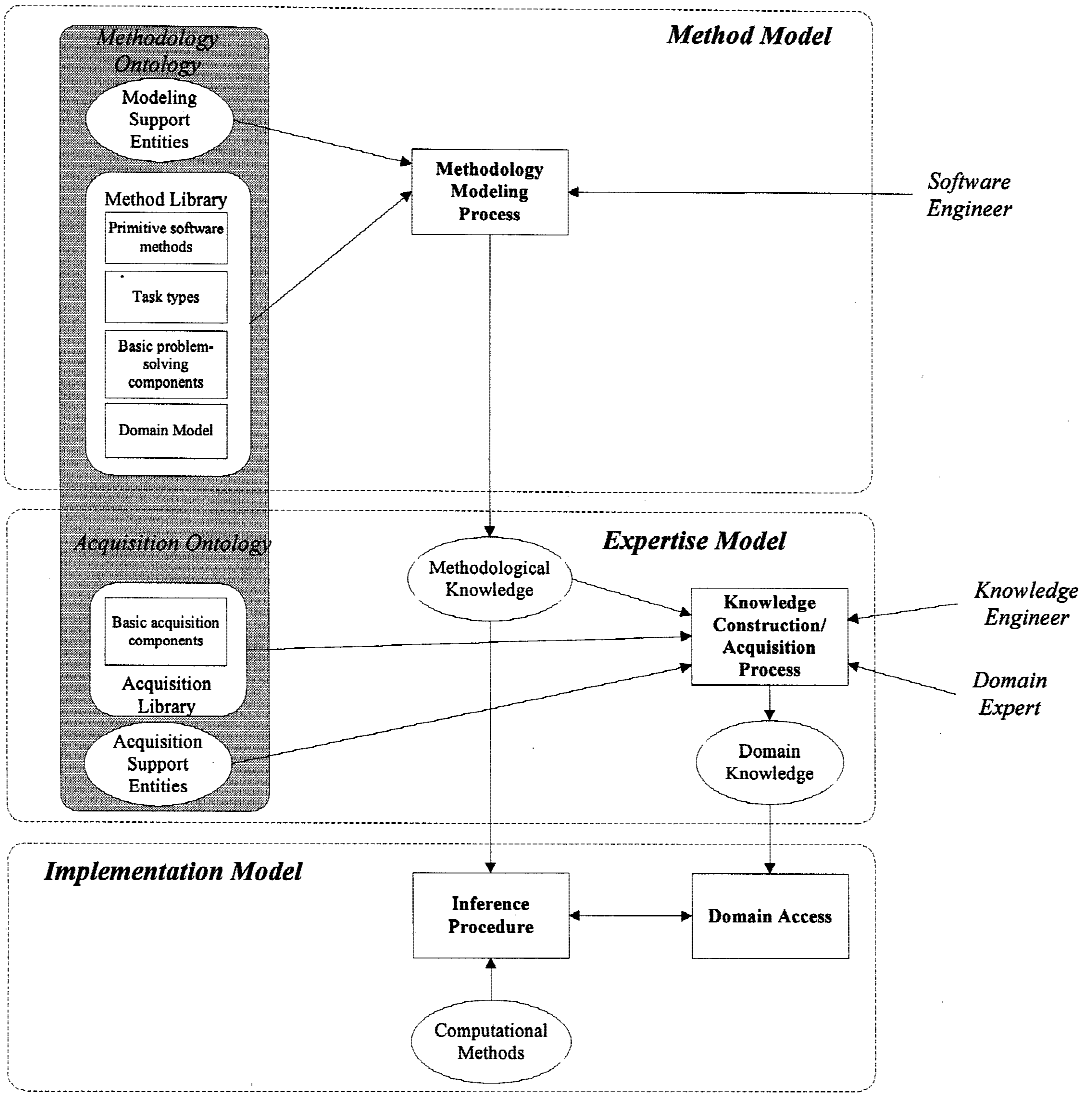


Fig. 1. Relationships among the method, expertise, and implementation models of GAMP.

Similarly, the major component of the expertise model is the *knowledge-acquisition ontology* (acquisition ontology for short), which contains a *knowledge-acquisition library* (acquisition library for short) and *knowledge-acquisition-support entities* (acquisition support entities for short). The acquisition library stores various knowledge-acquisition-related methods. The acquisition-support entities use the methodological knowledge derived by the method model to exert the acquisition library to do knowledge acquisition.

The rest of the paper will elaborate this architecture and demonstrate its feasibility by applying the approach to design an RA tool for network-management systems. Section 2 will first give a brief description about our application domain, that is, describing major properties of the equipment in a SONET network and related management functions. This will provide the reader with a background to follow the illustrated examples in the subsequent sections. Section 3 then introduces the philosophical basis of the approach, including the methodology and acquisition ontologies and their application. Section 4 exemplifies the application of this approach on the domain described in Section 2. Section 5 concludes the work and provides a discussion.

2. CHARACTERISTICS OF SONET NM S/W

2.1. SONET network

SONET (Synchronous Optical NETwork) is an international standard originated from the efforts of the RBHCs (Regional Bell Holding companies), Bellcore (Bell Communications Research), and MCI (Black & Waters, 1996). SONET

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

is an ANSI standard (ANSI T1.105, 1995); ITU has another similar standard—SDH (Synchronous Digital Hierarchy) (G.784, 1994). Before SONET, equipment from different manufacturers could not communicate with each other, making routing traffic between carriers a cumbersome task. The standard provides a common way of building and running fiber optic networks. Moreover, it also contains a new hierarchy for multiplexing and transmitting signals up to 2.4 Gbps. A SONET-based network thus offers more benefits, including cost reduction through network simplification, revenue increase from new services, improvement of network robustness, and survivability, and multivendor compatibility through open interface and compliance with the standard. One final advantage: SONET provides an additional channel to transmit management information, which allows the manager to better monitor, maintain, operate and evolve the network.

Chung-Hwa Telecommunications, the major Telco in Taiwan, deployed a SONET network into their long-distance backbone in 1995. It has been playing an important role in carrying vast voice and data signals on the island. It contains various types of equipment, including terminal multiplexers, add/drop multiplexers, digital-loop carriers, digital cross-connects, and regenerators (Lee et al., 1997), among which the add/drop multiplexers are called the building block of a SONET network. They are responsible for communicating with existing equipment and facilities through a DS-N (Digital-Signal-type-N) electrical interface, and play a key part in controlling traffic within a SONET network. Good network-management software is important in assuring the normal operation of the network. Our target is to develop a specific RA tool that can generate proper requirement specifications for the *A*dd/*D*rop *M*ultiplexer in 600 Mbps, or *ADM600*.

2.2. Architecture of network management software

For interoperability and extendibility, we followed the ISO standard (Klerer, 1988; ISO 7498-4, 1989) to design the network-management software architecture for ADM600 (Lin et al., 1991). In the standard, either a manager node or an agent node consists of *management application*, *management information base* (MIB), and OSI *common management information service entity* (CMISE). The management application is the active perspective of the network management, consisting of the following function areas: *fault*, *configuration*, *performance*, *security*, and *accounting management*. These function areas tell us what the network is doing, where everything is in the network, how the network is doing, who is using the network, and when the network is being used. The MIB is the passive perspective of the network management, consisting of all resources to be managed. They are referred to as the managed objects in the T1M1 generic network model (ANSI T1M1.5/90-010, 1990). These objects can be physical or logical types. Physical objects represent physical entities in the network; while logical objects support management requirements. CMISE refers to the services that manipulate communication requirement of the management activities (ISO 9595, 1990). They can be categorized into *data manipulation service*, *event reporting service*, or *direct control service*.

To realize these management functions in software, we first need to do RA. This is often done by a specific CASE tool, such as Structured Methodology for Excelerator II or Object-Oriented Design for Excelerator II (Budd & Woods, 1998), which, however, suffers several difficulties. First, most of these COTS (Commercial Of The Shelf) CASE tools are *surface-level* tools; they only help the user to edit, verify, and convert representation. They do not do deeper work, such as assisting the user in finding incompleteness, detect inconsistency, or suggest alternative designs. Although some RA tools mentioned above are fairly intelligent, they only work well under a specific software methodology within a specific domain. Once the domain type changes, they may not work well. Second, these tools do not support the reuse of the experience of previous RA episodes; we have to do RA from scratch. Finally, they do verification and checking only on the final product rather than on the process, which is preferred, however, because it facilitates the finding of errors in earlier stages and improves the correctness of the product. This has motivated us to develop the GAMP approach.

3. DESIGN PHILOSOPHY OF GAMP

GAMP is a generic ontology-based approach. It is generic in the sense that it enables the user to easily construct domain-specific RA tools for specific domains. It is ontology-based in the sense that it derives these specific RA tools from a unified ontological framework that structures software methodologies and knowledge-acquisition methods, so as to cope with integration granularity, sharing, and reuse of the methods. This design philosophy is well illustrated in Figure 1. The following sections will detail the two core ontologies and how they are used through the associated processes.

3.1. Methodology ontology

Based upon our conjecture that ontology is a shared logical specification that defines the ontological commitment to a particular conceptualization, this section now formally defines the methodology ontology.

First, we define $C_m = \langle D_m, W_m, \mathcal{R}_m \rangle$ to be the conceptualization of software methodology. $D_m = \langle D_m^m, D_m^s \rangle$, where D_m^m is a set of software methods that constitute a methodology, and D_m^s is a set of modeling strategies that support the modeling of methodological knowledge. W_m is a set of relevant states about the possible arrangement of the entities in D_m. $\mathcal{R}_m$ is a set of conceptual relations on the domain space $\langle D_m, W_m \rangle$.

Now, we define $O_M = \langle M, C_m, \mathcal{J}_m \rangle$ to be the methodology ontology that commits C_m. $M = \langle M^m, M^s, M^r \rangle$, where M^m

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

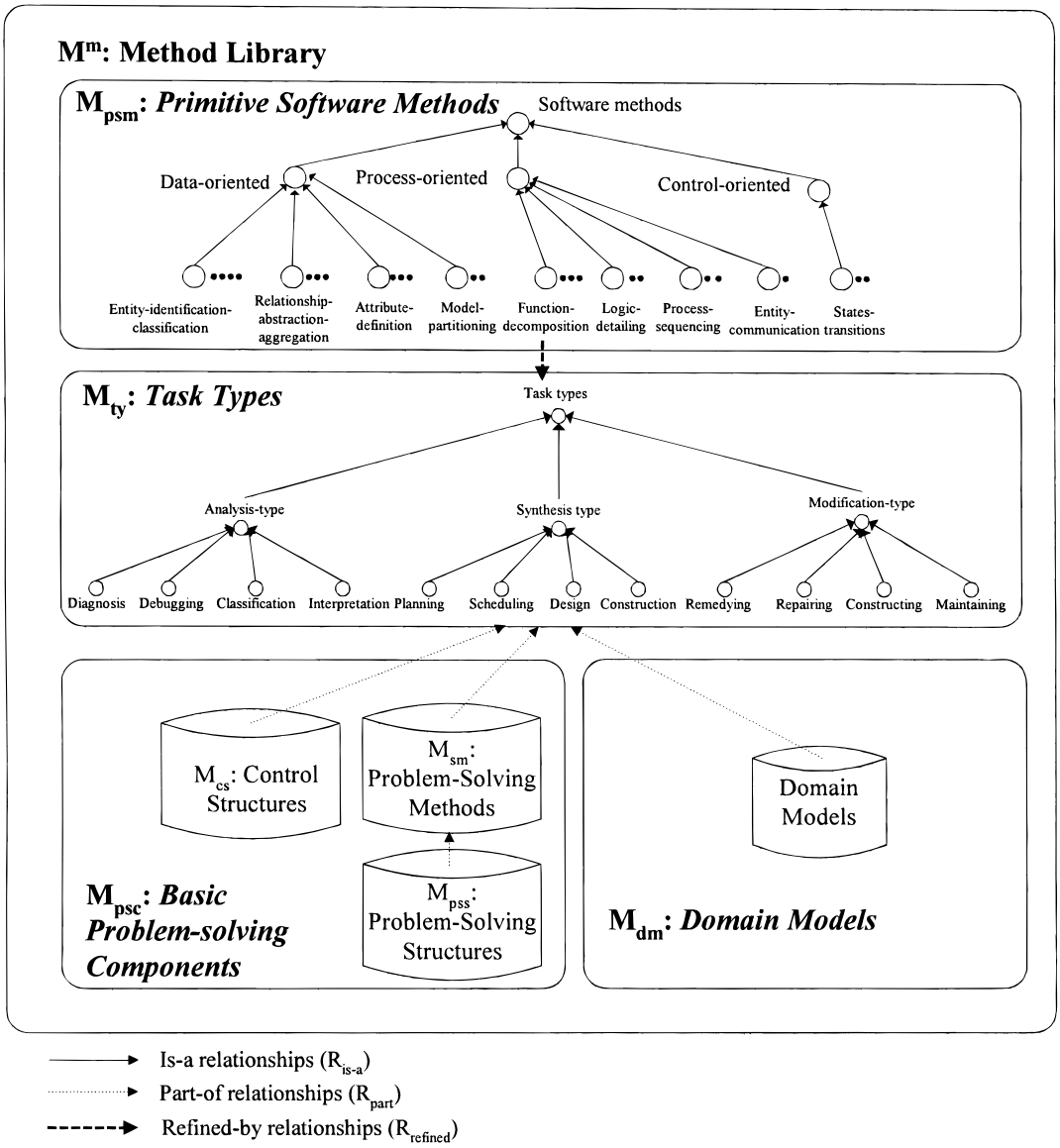


Fig. 2. M^m: Method library structure.

is a set of software-method symbols, and M^s is a set of modeling strategy symbols, and M^r is a set of relation symbols. $\mathcal{I}_m$: $M \rightarrow D_m \cup \mathcal{R}_m$ is an interpretation function assigning elements of D_m^m to the symbols of M^m, D_m^s to M^s, and $\mathcal{R}_m$ to M^r.

So far, we have given the methodology ontology a basic structure. We need further elaboration for each ingredient, starting from M^m. $M^m = \langle M_{psm}, M_{ty}, M_{psc}, M_{dm} \rangle$ is a software-methods structure, where M_{psm} is a set of primitive software methods; M_{ty} is a set of task types; $M_{psc}$is a set of problem-solving-related components; and $M_{dm}$is a set of domain models. Components in M_{psc} include M_{sm}, M_{cs}, and M_{pss}, where M_{sm} is a set of problem-solving methods, M_{cs} is a set of control structures, and M_{pss} is a set of problem-solving (decomposition) structures. Because M^m specifies the building blocks for constructing a software methodology, we thus collect them in a library and name it the method library as illustrated in Figure 2.

$M^s = \langle M_{ma}, M_{mf}, M_{mr} \rangle$ is a modeling-strategies structure, where M_{ma} is a set of methodology realities; M_{mf} is a set of abstract methodology features; and M_{mr} is a set of modeling rules. Because M^s specifies the strategies that support the construction of a software methodology, we term them modeling-support entities as illustrated in Figure 3.

Finally, $M^r = \langle R_{\text{map}}, R_{\text{input}}, R_{\text{operate}}, R_{\text{refined}}, R_{\text{is-a}}, R_{\text{part}} \rangle$ is a relationship structure, where R_{map} is a binary map-to relation from M_{ma} to M_{mf}; R_{input} is an input-to relation from M_{mf} to M_{mr}; R_{operate} is an operate-on relation from M_{mr} to M^m; R_{refined} is a refined-by relation from M_{psm} to M_{ty}; and $R_{\text{is-a}}$ is an is-a relation and R_{part} is a part-of relation specifying the taxonomy relationships among the elements of M^m. The arrows in Figures 2 and 3 illustrate M^r.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

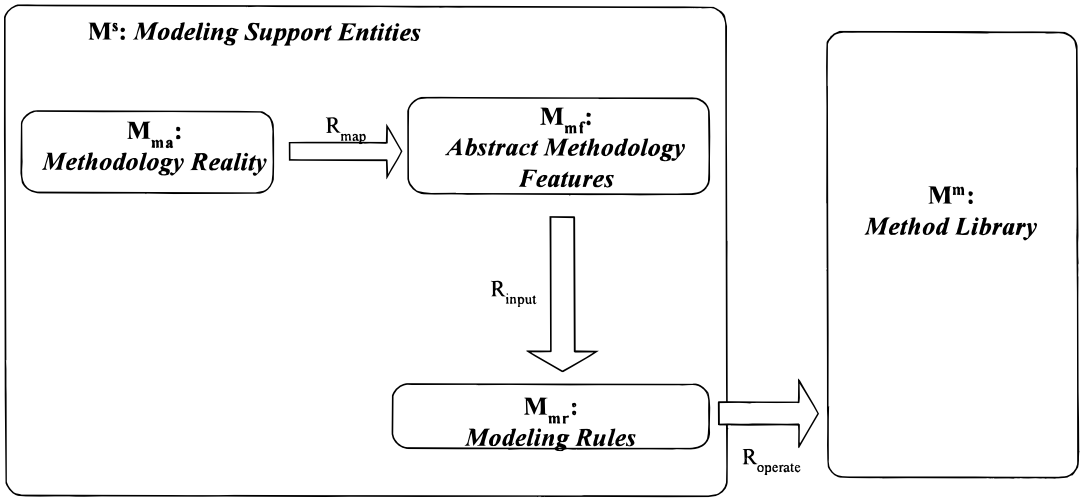


Fig. 3. M^s: Modeling support entities structure.

The basic meanings and structures of the methodology ontology are now completely defined. What is left is the definition of the attributes and structure of each ingredient. The following subsections refine them in detail, which also explains why these ingredients are selected to shape the methodology ontology.

3.1.1. Method library

According to the lexicology, "a methodology is a set of methods used for study or action in a particular subject" and "a *method* is a planned way of doing something." Putting this viewpoint into software engineering, a software methodology is composed of a set of software methods. This leads to the idea of *methodology decompilation*, which decompiles a methodology into *reusable* methods. Applying this idea allows us to derive primitive software methods from existing software methodologies (Short, 1991; David & Gretchen, 1992; Fichman & Kemerer, 1992; Bickerton & Siddiqi, 1993; Gomaa et al., 1996; Maiden & Rugg, 1996). These methods capture similar and essential notions in different methodologies in terms of a set of atomic actions. By properly synthesizing some of these methods, we can build a customized methodology.

The top-level of Figure 2 illustrates these concepts. Basically, nine types of primitive software methods are derived from the three most popular classes of methodology, that is, the *data-oriented* methodology, *process-oriented* methodology, and *control-oriented* methodology. They are *Entity-identification-classification*, *Relationship-abstraction-aggregation*, *Attribute-definition*, *Model-partitioning*, *Function-decomposition*, *Logic-detailing*, *Process-sequencing*, *Entity-communication*, and *State-transition*. Each method is further specialized to several particular primitive methods. For example, Entity-identification-classification has the following specialized primitive methods: *Entity-relationship-diagramming*, *Data-model-diagramming*, *Class-object-diagramming-layer1*, and *Information-structure-diagramming*. Figure 4 organizes the complete primitive software methods, that is, M_{psm}, using the $R_{\text{is-a}}$ relation into a hierarchy. The following illustrates an example primitive software method, called Class-object-diagramming-layer1#1, which uses the design task to identify the classes and objects for a problem.

Primitive-software-method	
class-name	Class-object-diagramming-layer1
instance-name	Class-object-diagramming-layer1 #1
subtype-of	Entity-identification-classification
goal-description	"Design a set of meaningful classes and objects for the system."
input	The set of initial terms for the problem statements
output	Final set of system classes and objects
has-task-knowledge	Design#1

The middle level of Figure 2 refers to the basic task types, M_{ty}. They are derived with the help of the MULTIS system (Tijerino et al., 1993; Mizoguchi et al., 1995) and our previous work (Chien & Ho, 1994). Basically, three generic task types are identified, that is, *Analysis-type*, *Synthesis-type*, and *Modification-type* (Chandrasekaran et al., 1992; Chandrasekaran, 1986). Each task type is then specialized; for example, *Diagnosis*, *Debugging*, *Classification*, and *Interpretation* tasks are specialized Analysis-type tasks. The major purpose of these tasks is to elaborate the above primitive software methods into proper problem-solving actions. For example, in Figure 2 because Class-object-diagramming-layer1 method is to "design" classes, objects, and their relationships, we can choose the "design" task to elaborate it. We use the R_{refined} relation to link the method

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

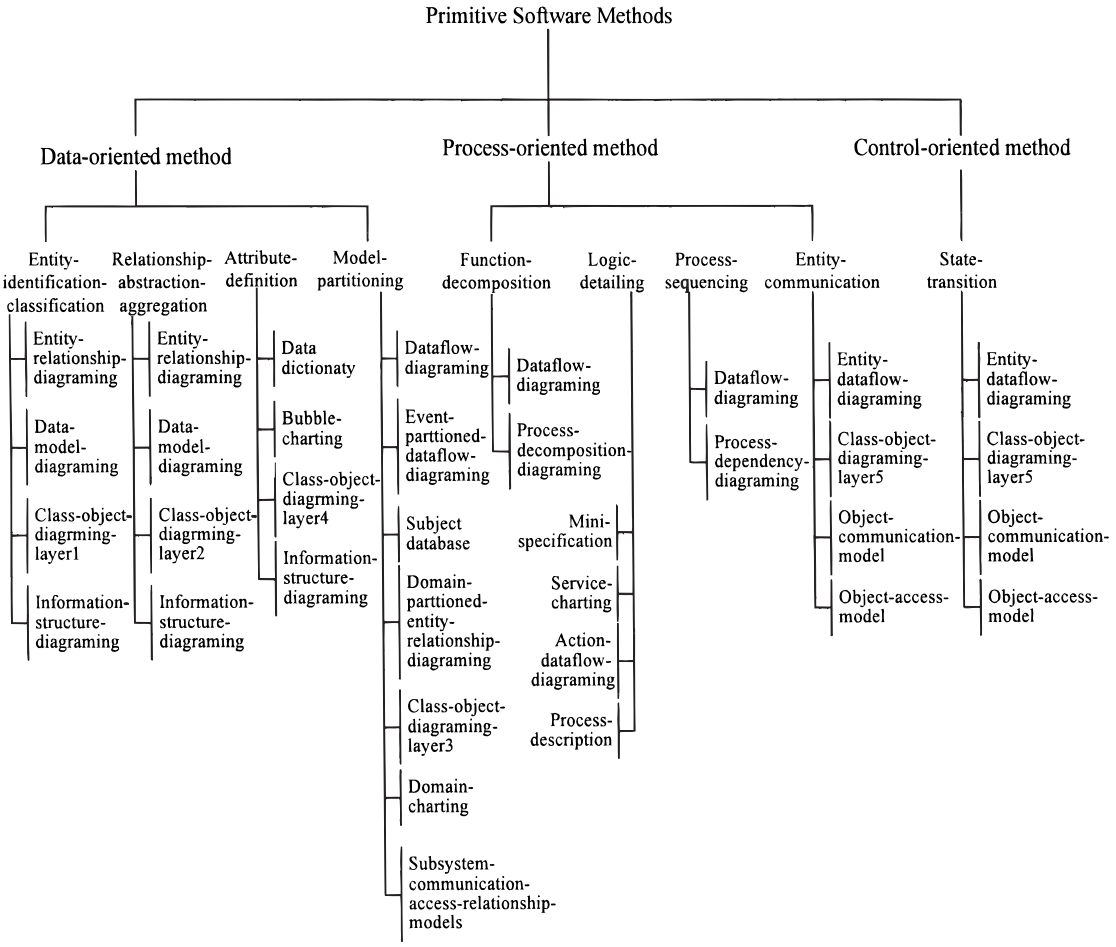


Fig. 4. Hierarchy of primitive software methods.

to the task. The following illustrates an example task type, called Parametric-design#1, which uses the Proposing-checking-revising#1 problem-solving method to find a design that satisfies a set of constrains.

Task	
class-name	Parametric-design
instance-name	Parametric-design#1
sub-type-of	Design
goal-description	"This function is intended to find a design that satisfies a set of constraints."
input	Parameters, Constraints, Revisions
output	Result-parameters
has-ps-knowledge	Proposing-checking-revising#1
has-control-knowledge	Propose-criticize-modify
has-domain-knowledge	Anchoring, Traveling; Constraining; Fixing

The above task structure naturally brings to light M_{psc}, the problem-solving components, and M_{dm}, the domain models. They are illustrated at the bottom level of Figure 2 and serves as the basic building blocks in the method library. First, each basic problem-solving component may contain a set of *problem-solving methods* (M_{sm}) along with two structures, that is, *problem-solving structures* (M_{pss}) and *control structures* (M_{cs}). They are made small enough to provide desired combination flexibility in representing the upper-level tasks. Specifically, the problem-solving structure decomposes a problem-solving method at a certain level into a set of lower-level problem-solving methods. It also specifies the input/output dependencies among the decomposed problem-solving methods. The idea here is that a problem-solving method at a sufficiently low level (possibly after several decompositions) can be subjected to straightforward implementation. The control structure imposes control on a problem-solving method, that is, it specifies its execution logic. Different control structures can be imposed on the same problem-solving method to demonstrate different problem-solving behavior, which is the reason why we choose to represent control structures as an independent library element. The following illustrates the example problem-solving method, that is, Proposing-checking-revising#1, used in the above task type.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

Problem-solving method	
class-name	Proposing-checking-revising
instance-name	Proposing-checking-revising#1
sub-type-of	Heuristic-construction
input	Parameter-set, Calculation-formulae, Constraint-set, Fix-set
output	final set of assigned parameters
goal-description	"This method is intended to find a artifact that satisfies a set of constraints."
assumption	(≠ Parameter-set *nil*)

The following exemplifies a problem-solving structure, named P&V-structure, to decompose the above Proposing-checking-revising#1 method into three constituent methods, that is, Proposing, Checking, and Revising.

Problem-solving-structure	
name	P&V-structure
problem-solving method	Proposing-checking-revising #1
constituents	Proposing, Checking, Revising
input-output-dependencies	(Parameter-set,Calculation-formulae → Parameters), (Parameters,Constraint-set → Violation), (Violation, Fix-set → Parameters)
description	"This structure decomposes the Proposing-checking-revising method."

The following demonstrates a control-structure, named Propose-criticize-modify, that specifies the execution logic for the above three constituent methods.

Control-structure	
name	Propose-criticize-modify
problem-solving method	Proposing-checking-revising#1
input-role	Parameter-set, Calculation-formulae, Constraint-set, Fix-set
working-role	Parameters
output-role	Parameters, Violation
control-logic	Init (Parameter-set) Repeat *Proposing* (Parameter-set, Calculation-formulae, Parameters) *Checking* (Parameters, Constraint-set, Violation) *Revision* (Violation, Fix-set, Parameters) Until (= Parameter-set *nil*)

Finally, the domain model, M_{dm}, represents the general structural knowledge about the domain. We use it to materialize the input and output slots of a task. Some typical models are the *causal* data model that represents causal-dependency relationships, for example, those between symptoms and diseases in a medical system; the *structural* data model that specifies structure-hierarchy relationships, such as the connections of the physical devices in a telecommunication-network-management system, etc. Formal definitions of these domain models can be found elsewhere (Chien & Ho, 1994).

3.1.2. Modeling support entities

Recall that Figure 3 elaborates the modeling support entities structure, M^s. In the figure, we first introduce the methodology reality, M_{ma}, to capture the important features of a software system developed using a particular software methodology because it provides important clues in selecting primitive software methods for constructing proper methodological knowledge. However, a direct use of its features to select primitive software methods would be unrealistic, because each methodology tends to define its own specific terminology. Abstraction of the methodology features from different terminology becomes a must; thus, we need to define *abstract* methodology features.

By surveying the works in the literature regarding the possible features of a software system (Jorgensen, 1986; Wang et al., 1989; Destor et al., 1990; Guenther & Wackerbarth, 1993; Lu et al., 1994), we have proposed *purpose*, *function*, *data*, and *domain* as four major dimensions for analyzing features and accordingly proposed a hierarchy of abstract methodology features, M_{mf} (Lin & Ho, 1998*a*). Figure 5 illustrates this hierarchy. We have also defined the canonical structure for the features. We exemplify an abstract methodology feature below. This feature determines whether the purpose of a system is strategic or not.

Methodology-feature	
class-name	System-goal
instance-name	Strategic
sub-type-of	Purpose-feature
feature-value	yes
value-constraint	yes/no
question-prompt	"Does the system conduct prediction or judgment using massive data from historical or external sources?"
comment	"This type of system conducts prediction or judgment using massive data from historical or external sources."

Finally, the modeling rules, M_{mr}, are introduced to specify the associations between the abstract methodology features and the method library components. This implies that the method library components will be identified and accessed through the modeling rules. Basically, a modeling rule gives as "output" an operation (e.g., select, combine, etc.) and the library components that the operation can work on (e.g., primitive software methods, task types, etc.). In other words, its IF part contains the description of one or more methodology features, while the THEN part contains the operations and the corresponding library components. To capture the associations at various levels of

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

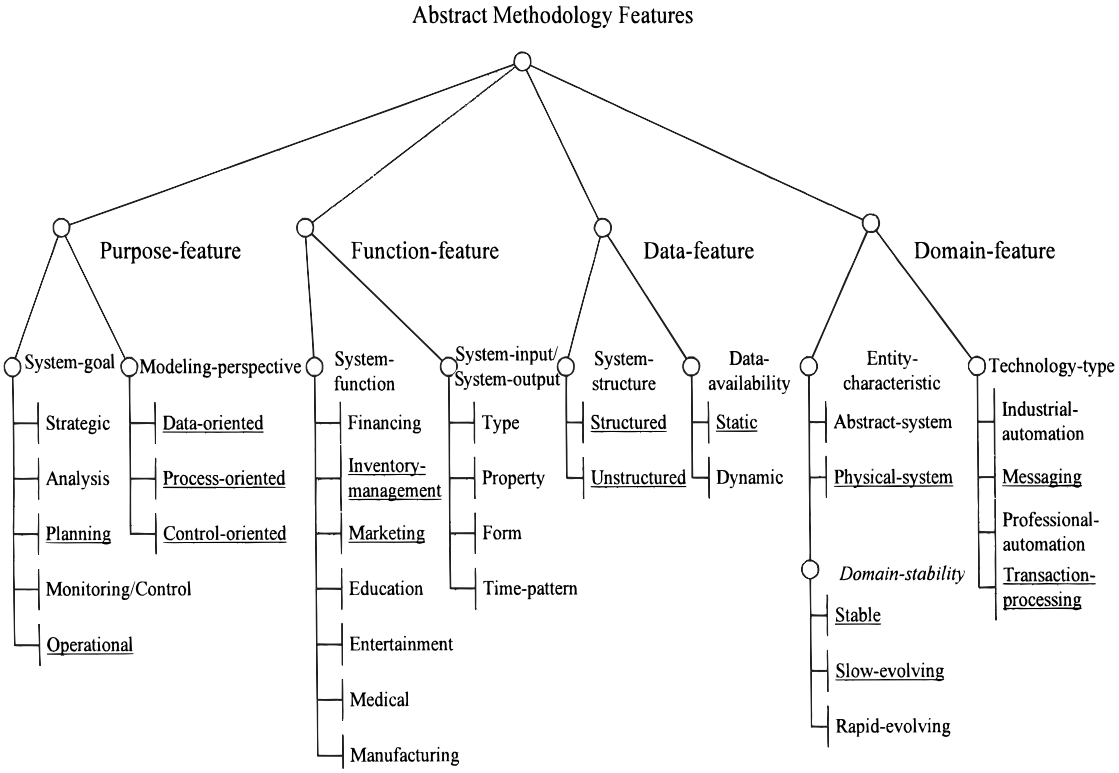


Fig. 5. Hierarchy of abstract methodology features.

abstraction, we also structure the rules in a hierarchy. Figure 6 illustrates such a hierarchy constructed around the operations suggested by the modeling rules. It contains four types of operations at the first layer, that is, *selection-type*, *combination-type*, *specialization-type*, and *generalization-type*. The selection-type selects appropriate primitive software methods, task types, problem-solving methods, problem-solving structures, control structures, or domain

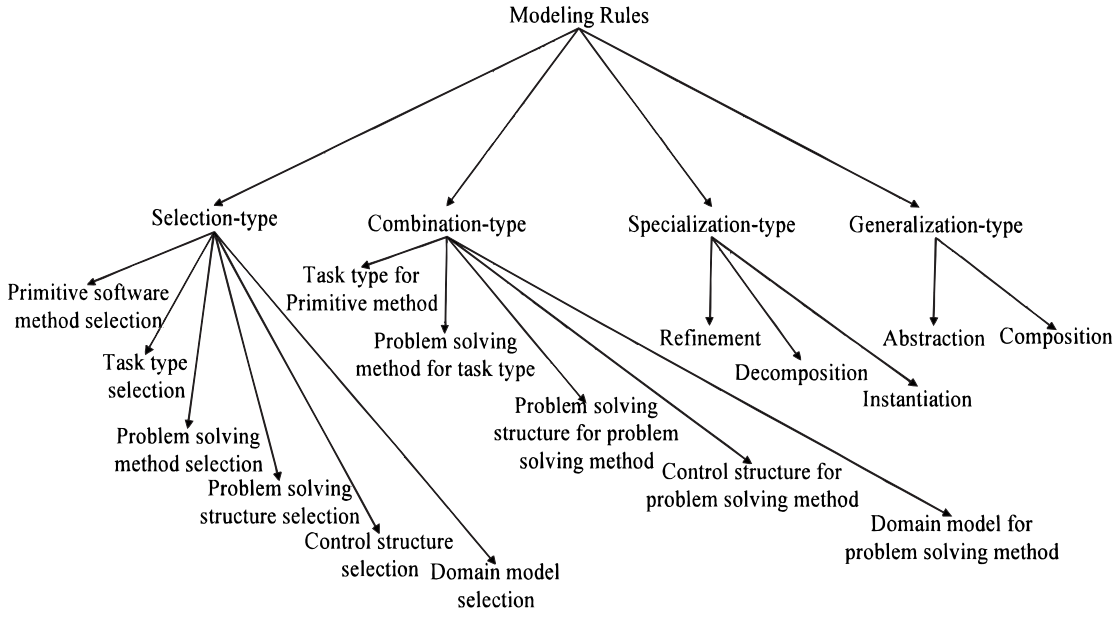


Fig. 6. Type hierarchy of modeling rules.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

models. The combination-type aggregates constituent components. The specialization-type elaborates a method, task, or component. The generalization-type generalizes ground-instance components (Ouyang, 1998). Example operations in the modeling rules include selecting a library component, instantiating a primitive software method, and refining a primitive software method. More specific operations include selecting a primitive software method, assembling a method from basic problem-solving components, selecting a task for a primitive software method, applying a problem-solving method to a task, and selecting a problem-solving structure for a problem-solving method.

A selection-type modeling rule is exemplified below. It guides the user to select the "Entity-identification-classification" primitive software method based on the following abstract methodology feature-value pairs: (Monitoring/Control Yes), (Messaging Yes), (Structured Yes), and (Dynamic Yes).

Modeling-rule	
class-name	Primitive-software-method-selection
instance-name	Primitive-software-method-selection #1
sub-type-of	Selection-type
methodology-feature	(Monitoring/Control Yes)(Messaging Yes) (Structured Yes)(Dynamic Yes)
method-model-state	*nil*
operation	Selection
library-component	Entity-identification-classification

3.2. Acquisition ontology

The acquisition ontology can be formally defined similarly to that for the methodology ontology. Let us start with the conceptualization of knowledge acquisition. We define $C_a = \langle D_a, W_a, \mathcal{R}_a \rangle$ to be the conceptualization of knowledge acquisition. $D_a = \langle D_a^m, D_a^s \rangle$, where D_a^m is a set of knowledge acquisition methods that acquire knowledge, and D_a^s is a set of knowledge acquisition strategies that support the acquisition of domain knowledge. W_a is a set of relevant states about the possible arrangement of the entities in D_a. $\mathcal{R}_a$ is a set of conceptual relations on the domain space $\langle D_a, W_a \rangle$.

Then, we define $O_A = \langle A, C_a, \mathcal{I}_a \rangle$ to be the acquisition ontology that commits C_a. $A = \langle A^m, A^s, A^r \rangle$, where A^m is a set of knowledge-acquisition-method symbols, and A^s is a set of knowledge-acquisition-strategy symbols, and A^r is a set of relation symbols. $\mathcal{I}_a$: $A \rightarrow D_a \cup \mathcal{R}_a$ is an interpretation function assigning elements of D_a^m to the type symbols of A^m, D_a^s to A^s, and $\mathcal{R}_a$ to A^r.

Given these basic structure definitions, we now elaborate each ingredient below. First, $A^m = \langle A_{am}, A_{im}, A_{ds} \rangle$ is a knowledge-acquisition structure, where A_{am} is a set of acquisition methods; A_{im} is a set of interaction methods; and A_{ds} is a set of domain symbols. Because A^m specifies the building blocks for acquiring domain knowledge, we collect them in a library and name it the acquisition library as illustrated in Figure 7.

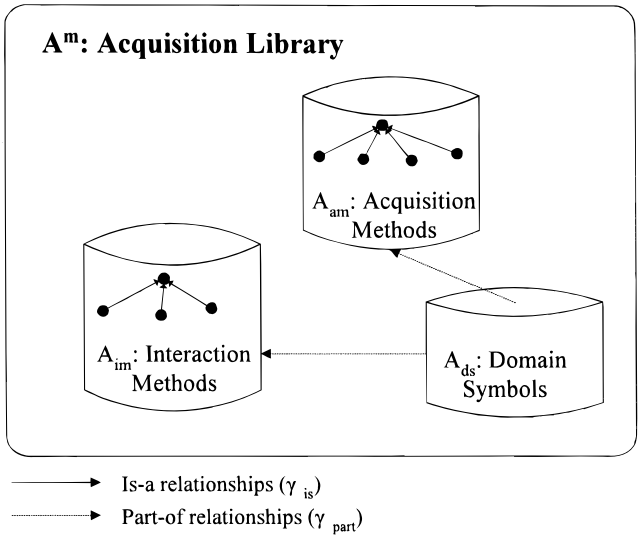


Fig. 7. A^m: Acquisition library structure.

$A^s = \langle A_{cf}, A_{cr}, A_{ar} \rangle$ is a knowledge-acquisition-strategies structure, where A_{cf} is a set of abstract construction features; A_{cr} is a set of construction rules; and A_{ar} is a set of acquisition rules. Because A^s specifies the strategies that support the acquisition of domain knowledge, we term them acquisition-support entities as illustrated in Figure 8.

Finally, $A^r = \langle \gamma_{\text{input}}, \gamma_{\text{operate}}, \gamma_{\text{is-a}}, \gamma_{\text{part}} \rangle$ is a relationship structure, where γ_{input} is a binary input-to relation from A_{cf} to A_{cr}; γ_{operate} is an operate-on relation from A_{cr} to A^m or form A_{ar} to A^m; $\gamma_{\text{is-a}}$ is an is-a relation and γ_{part} is a part-of relation specifying the taxonomy relationships among the elements of A^m. The arrows in Figures 7 and 8 illustrate A^r.

The basic meanings and structures of the acquisition ontology are now completely defined. The following subsections go on to define the attributes and structure of each ingredient, which also explains why these ingredients are selected to shape the acquisition ontology.

3.2.1. Acquisition library

First, with the knowledge of problem-solving methods, M_{sm}, and domain models, M_{dm} as defined in the methodology library, we can define a set of acquisition methods, A_{am}, to acquire correct domain knowledge to be represented by the domain models and to be used by the problem-solving methods. This is because by analyzing the problem-solving methods and the domain models, we not only know what types of knowledge are needed but we also know what models are appropriate for representing the knowledge. For example, we can define an acquisition method called *Constraint-proposing* to correspond to problem-solving method *Checking*. This acquisition method knows to acquire constraints, since Checking method needs that for problem solving. The following

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

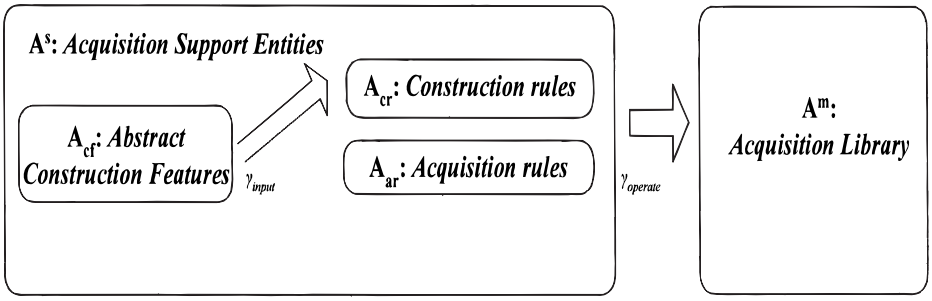


Fig. 8. A^s: Acquisition support entities structure.

illustrates the Constraint-proposing#1 example acquisition method.

Acquisition-method	
class-name	Constraint-proposing
instance-name	Constraint-proposing#1
description	"Acquiring design constraints and their values."
has-domain-symbol	Equipment-constraint; Agent-constraint

The construction rules, A_{cr}, guide the knowledge engineer in selecting a proper acquisition method for a given domain. They also guide him in delivering what concepts he thinks are most significant in the domain into the selected acquisition methods. The customized acquisition methods then know what types of domain knowledge to acquire from the domain experts. We represent these significant concepts by the domain symbols, A_{ds}.

For the domain expert to articulate domain knowledge in a natural and effective way, proper interaction methods, A_{im}, are important. We have conducted a survey on manual KA methods and interactive KA tools (Boose & Gaines, 1988; Marcus, 1988; Boose, 1989; Kitto & Boose, 1990) and identified four basic interaction techniques: *grid-driven procedure*, *menu-driven procedure*, *table-driven procedure*, and *question/answer procedure* (Chien & Ho, 1994). Basically, the grid-driven procedure is used in the repertory grid method, which elicits and analyzes the domain expert's world model based on personal construct theory and implemented in ETS (Boose, 1984) and AQUINAS (Boose, 1990). The menu-driven procedure is used in the process trace method, which traces the decision-making process of the domain experts and unfolds their problem-solving strategies. It, coupled with the table-driven procedure, was implemented in MOLE (Eshelman, 1988), MORE (Kahn et al., 1985), and SALT (Marcus & McDermott, 1989) to select menus and propose parameters. The question/answer procedure conducts structured/unstructured interviews, used in ROGET (Bennett, 1985) and SIS (Kawaguchi et al., 1991). The following illustrates an example interaction method, called Table-driven#1.

Interaction-method	
class-name	Table-driven
instance-name	Table-driven#1
description	"Using a table to do interaction."
has-domain-symbol	Equipment
symbol-attributes	operation-state, alarm-status
value-constraints	characters, Characters

Depending on what interaction method works best for a domain symbol, the construction rules can guide the knowledge engineer to construct a specific interaction method for the domain symbol by storing its attributes in the interaction method. Because the domain symbol is contained in the customized acquisition method, using it as a link to the interaction method empowers each acquisition method with a proper interaction method for the contained domain symbol.

3.2.2. Acquisition support entities

First, we treat domain knowledge acquisition as a two-process event for system flexibility and reusability (Chien & Ho, 1994). The first process interacts with the knowledge engineer to construct a *knowledge-acquisition structure* (acquisition structure for short), while the second process uses the structure to interact with the domain expert to perform knowledge acquisition. Accordingly, we define two types of acquisition support entities, namely, the construction rules, A_{cr}, to support the first process and the acquisition rules, A_{ar}, to support the second process.

Similar to the work on the method model, we have defined a set of abstract construction features, A_{cf}, to capture the characteristics of a domain with the intention to help the knowledge engineer select adequate components from the acquisition library to construct an acquisition structure. By surveying the literature (Gaines & Boose, 1988; McGraw & Harbison-Briggs, 1989; Boose & Gaines, 1990; Marcus, 1990), we propose *knowledge*, *domain*, and *interface* as the three major dimensions for analyzing features and accordingly proposed a hierarchy of abstract construction features (Lin & Ho, 1996). The construction rules, A_{cr}, then use these features to identify basic acquisition components. Similar to the modeling rules, a construction rule produces as "out-

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

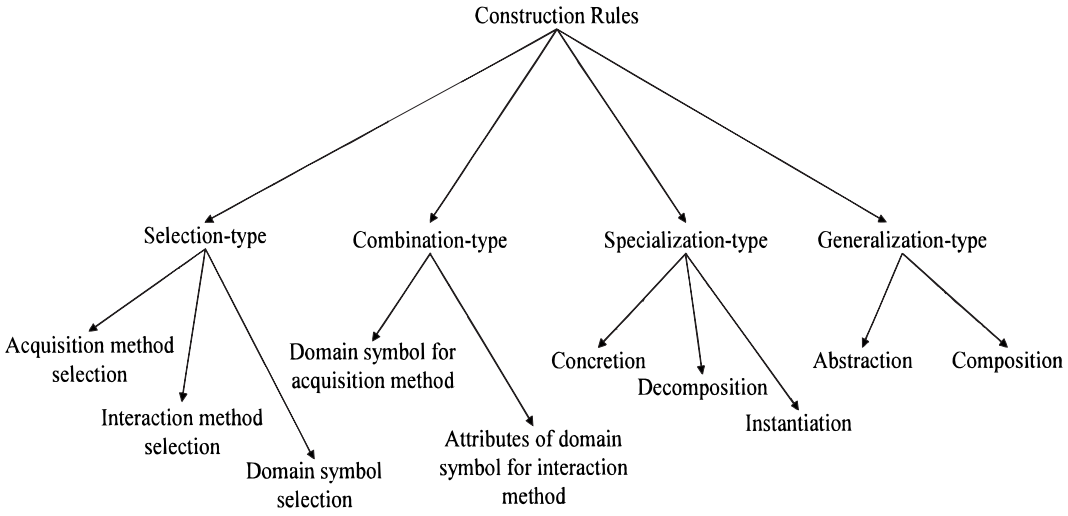


Fig. 9. Type hierarchy of construction rules.

put" an operation (e.g., select, combine, etc.) and the type of the acquisition components that the operation will work on (e.g., acquisition methods, interaction methods, and domain symbols).

Figure 9 shows the hierarchy of the construction rules, which reflects the associations at various levels of abstractions according to their operations. The first layer contains four types of operations, that is, *selection-type*, *combination-type*, *specialization-type*, and *generalization-type*. The selection-type operations select appropriate acquisition methods, interaction methods, or domain symbols. The combination-type operations aggregate constituent domain symbols for acquisition methods or interaction methods. The specialization-type operations instantiate interaction methods or domain symbols with values. The generalization-type generalizes ground-instance components. A selection-type construction rule is exemplified below. It guides the knowledge engineer to select the "Parameter-proposing" acquisition method based on some construction features.

Construction-rule	
class-name	Acquisition-method-selection
instance-name	Acquisition-method-selection#1
sub-type-of	Selection-type
construction-feature	((Taxonomic Yes)|(Episodic Yes)) (Object Yes)
method-model-state	(Problem-solving-method proposing)
expertise-model-state	*nil*
operation	Selection
library-component	Parameter-proposing

Finally, the acquisition rules, A_{ar}, interpret the generated acquisition structures to acquire domain knowledge. We have structured the acquisition rules into a type hierarchy (Fig. 10). It contains two types of operations at the first layer, that is, *selection-type* and *specialization-type*. The selection-type operations select two types of entities: *acquisition methods* and *interaction methods*. The *acquisition-method-selection-type* rules are specialized into 12 subtypes in accordance with the acquisition methods they select. The *interaction-method-selection-type* rules are categorized into two types, that is, domain-symbol type and attribute type too. The specialization-type rules are responsible for eliciting values for the domain symbols according to the interaction methods associated with the symbols. A selection-type acquisition rule is exemplified below. It selects all acquisition methods of class "Parameter-proposing," and finds all domain symbols and their attributes contained in these acquisitions methods.

Acquisition-rule	
class-name	Parameter-proposing-type-selection
instance-name	Parameter-proposing-type-selection#1
subtype-of	Acquisition-method-selection-type
expertise-model-state	(Acquisition-method Parameter-proposing)
operation	Selection
library-component	(Acquisition-method.class-name = Parameter-proposing)

3.3. Application of the ontologies

We have developed a reasoning architecture similar to SOAR (Laird et al., 1987; Rosenbloom et al., 1993) to apply the ontologies. It is designed as a general-purpose mechanism to produce methodological knowledge and domain knowledge. Basically, it contains an *inference engine* that applies the *knowledge bases* to develop target knowledge on a *working space*. There are two knowledge bases in the architecture. One is called *rule base*, storing the support entities;

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

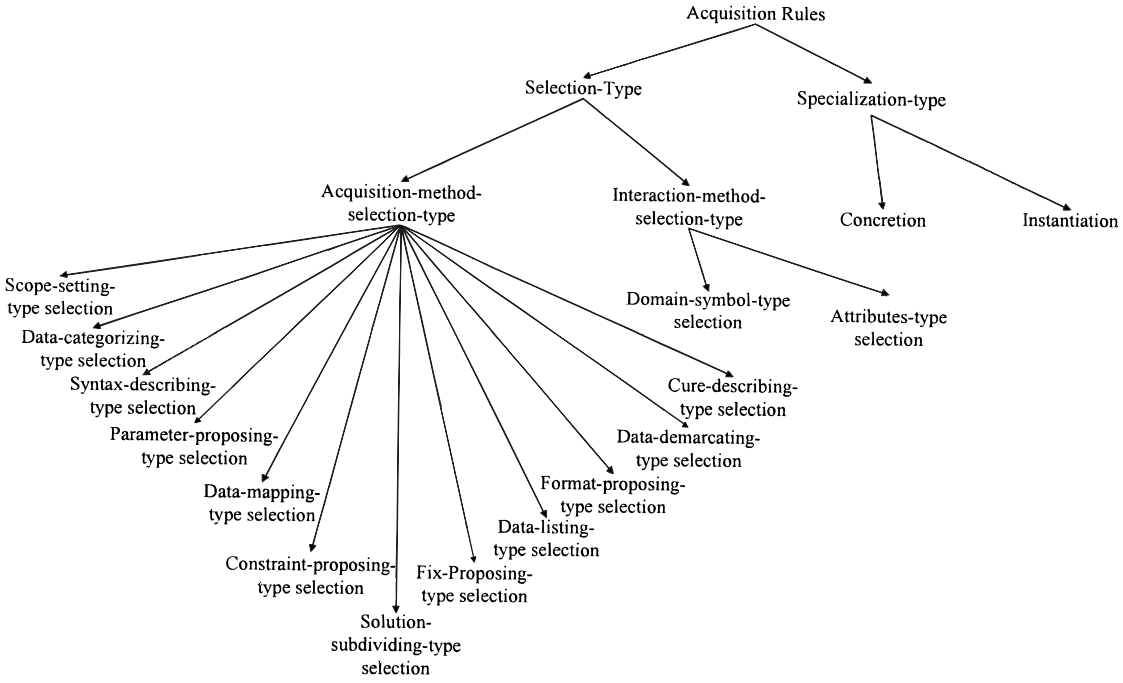


Fig. 10. Type hierarchy of acquisition rules.

the other is called *component base*, storing the library components. The support entities refer to the modeling rules (M_{mr}), construction rules (A_{cr}), or acquisition rules (A_{ar}); while the library refers to the method library (M^m) or acquisition library (A^m). The inference engine consists of an *agenda* and an *arbitrator*. The agenda classifies the invoked support entities. The arbitrator uses a *recognize-act* cycle (Brownston et al., 1985) to control reasoning. The working space contains a *working memory* and a *working area*. The latter stores the evolving desired output, for example, methodological knowledge. The former stores the current status. An *Asker* is equipped as an interface to get relevant feature values from the user.

3.3.1. Knowledge bases

To reflect the semantics of the ontologies to the control strategy of the reasoning architecture, the support entities in the rule base are properly prioritized in accordance with the semantic constraints and the structural dependency relationships in the ontologies. Because the operation types of the support entities and the library components with their relationships bear these semantics, the priority assignment is done accordingly. First, there are two major operation types, namely, selection type and nonselection-type (i.e., combination type, specialization type, and generalization type). The selection-type rules are set with the highest execution priority because the selection of an initial method structure is always the first thing to do. Not all selection-type rules are of the same priority, however. They are designated different priorities according to the components of library on which they operate. Thus, in the modeling rules the *primitive-software-method-selection* rules have a higher priority than the *task-types-selection* rules, which are in turn higher than the *problem-solving-method-selection* rules, and so on. Similarly, in the construction rules the *acquisition-method-selection* rules have a higher priority than the *interaction-method-selection* rules, which are in turn higher than the *domain-symbol-selection* rules. Finally, in the acquisition rules the *acquisition-method-selection-type* rules are designated the highest priority than the other types. Furthermore, within the subclasses of the acquisition-method-selection-type rules, the *scope-setting-type-selection* rules have a higher priority than the *data-categorizing-type-selection* rules, which are in turn higher than the *parameter-proposing-type-selection* rules, and so on.

Thus, the knowledge base not only contains rules and components but also contains procedure-control knowledge. As a matter of fact, it involves a variety of physical semantics and constraints too. To guarantee that the system can work properly, a representation mechanism that is expressive enough to capture as many types of knowledge as possible and is still subject to formal analysis is very important (Chander et al., 1997). We have developed a net-based representation technique for this (Lin & Ho, 1998*b*). The verification, validation, and refinement techniques on the knowledge represented by the mechanism can be found in

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

Chu (1995), which has done a comprehensive survey on state-of-the-art techniques on the evaluation of knowledge bases, including Kiper (1992).

3.3.2. *Inference engine*

The agenda of the inference engine keeps the invoked support entities in two boards. The *backward-inference board* stores the invoked selection-type rules, while the *forward-inference board* stores the other types of invoked rules. The backward-inference board has a higher execution priority than the forward-inference board. The arbitrator applies a two-layer procedure to control the reasoning process as illustrated in Figure 11. At the upper-layer, the arbitrator scans the rules to select the highest-priority rules whose conditions are partly matched by the features or model-states (called patterns) in the working memory, and places them in the appropriate board. These rules are called *probable* rules. It then performs a recognize-act cycle as the lower-layer procedure on the agenda board. First, the patterns in the working memory are matched against the feature and

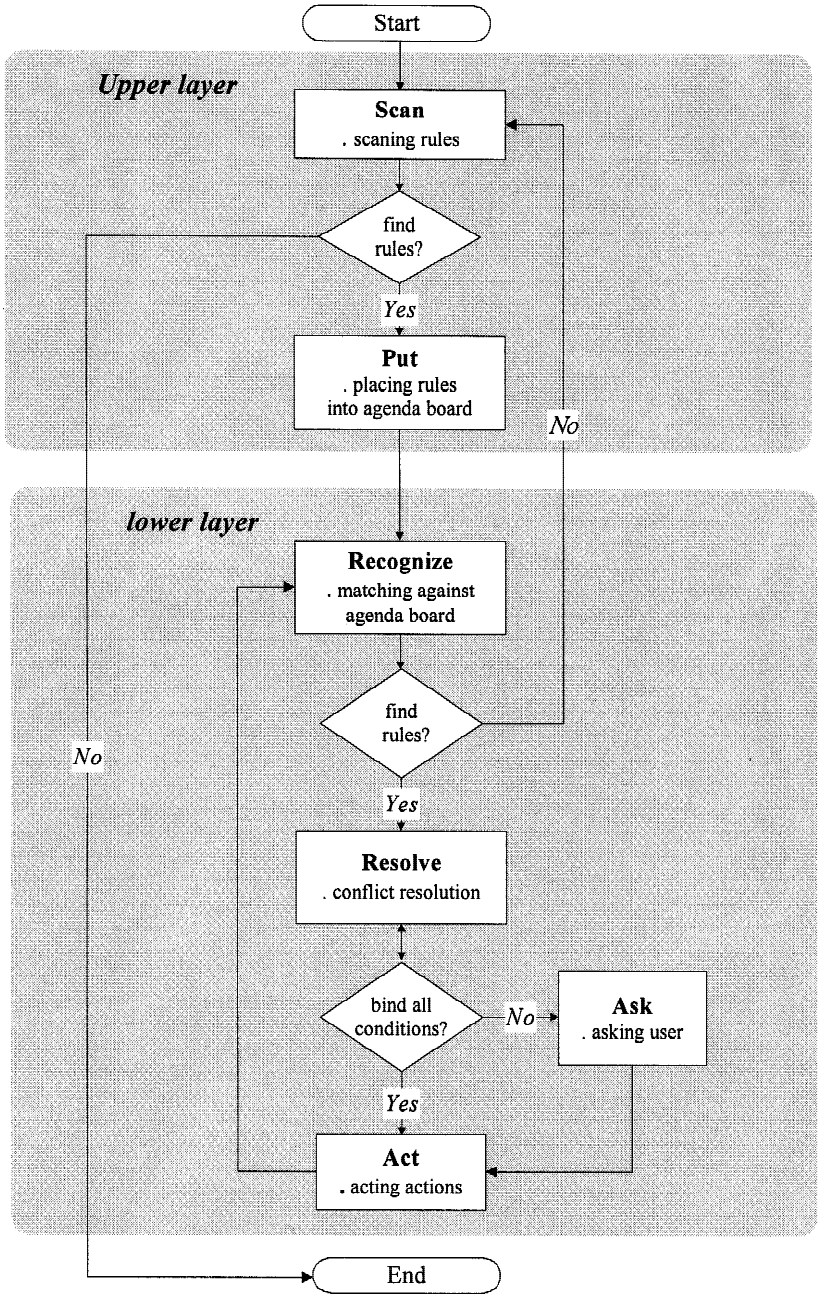


Fig. 11. Reasoning procedure of the architecture.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

the model-state slots of the probable rules in the agenda. This will recognize a subset of rules, called *conflict set*, which contains partially matched slots. The rules in the conflict set are said to be *enabled*. One of the rules in the conflict set is then selected by the conflict resolution strategy based on the number of the matched patterns. This selected rule is said to be *triggered*. If the triggered rule cannot be executed due to lack of more feature values, the arbitrator will call the Asker to get proper values. It will be executed when all its feature values are provided. (The rule is said to be *fired*.) Executing the action of a rule will change the contents of the working space. After the firing, the arbitrator repeats the lower-layer cycle until no other rules can be enabled. It will now go to execute the upper-layer procedure again to scan next probable rules into the agenda, and to start another lower-layer cycle. The upper-layer cycle stops when the arbitrator cannot find new probable rules. In summary, the two-layer procedure along with the prioritized rule-base forms the basic control strategy for the reasoning architecture. The conflict resolution process for the lower-layer procedure is then done by checking the number of the matched patterns in the condition part of the conflicting rules. To make it flexible, the system also allows the designer to associate general strategies, such as preference with the rules for conflict resolution.

Two reasons justify why we adopt this two-layer procedure to implement the reasoning process. One, this procedure correctly reflects the semantics of the ontologies through the prioritized rule base. Two, it reduces the number of rules recognition by scanning them all into the agenda at a time (at the upper-layer), which speeds up the reasoning process. For simplicity, we treat this reasoning architecture as containing the following operations: *Scan*, *Put*, *Recognize*, *Resolve*, *Ask*, and *Act*. *Scan* scans the support entities to find the most probable rules. *Put* places these rules in proper boards according their types. *Recognize* matches them with the patterns in the working memory to find enabled rules into a conflict set. *Resolve* adopts conflict resolution strategies to select a triggered rule from the conflict set. *Ask* is responsible for asking users for values for the triggered rule. Finally, *Act* executes (fires) the rule and changes the content of the working space.

3.3.3. User interface

Inside the user interface module is an *inventory* that contains the abstract methodology/construction features (M_{mf}, A_{cf}), as well as their corresponding questions. The Asker module uses the inventory to ask the user about the value of a methodology/construction feature and returns it to the arbitrator for further reasoning.

3.3.4. RA-tool-generation processes

First, the reasoning architecture works as a *methodology-modeling process* (Fig. 1) when it uses the methodology ontology to generate methodological knowledge (Fig. 12). The figure shows that the modeling rules are placed in the rule base according to their priorities, while the method library is placed in the component base. The inference engine applies these two knowledge bases to derive methodological knowledge on the working space. The agenda classifies the invoked modeling rules. The arbitrator applies the two-layer reasoning procedure and follows the rule priorities to control reasoning. The methodology features are placed in the inventory within the user interface. The working memory stores the current status, including problem-instance-specific data, facts, and intermediate control information, for example, values of abstract methodology features, method model states, etc. The working area stores the evolved software methods (methodological knowledge).

The reasoning architecture works as a *knowledge-construction process* (Fig. 1) by using the acquisition ontology to construct acquisition structures (Fig. 13). The construction rules are placed in the rule base according to their priority. The acquisition library is placed in the component base. The construction features are placed in the inventory inside the user interface. The working memory stores the current reasoning status including domain-instance-specific data, facts, and intermediate control information, for example, values of abstract construction features, method-model-state, and expertise-model-state, etc. The working area contains the evolved acquisition structure, the desired output.

Finally, the reasoning architecture works as a *knowledge-acquisition process* (Fig. 1), which conducts knowledge acquisition using the acquisition structures just developed (Fig. 14). The acquisition rules are placed in the rule base according to their priority. The component base is no longer needed. The working memory stores the current reasoning status including data and expertise-model-state (acquisition structure). The working area contains the evolved domain knowledge.

4. APPLICATION: GENERATION OF AN RA TOOL FOR NETWORK MANAGEMENT

This section illustrates how the approach is used to generate an RA tool for the domain mentioned in Section 2. Specifically, we will produce specific methodological knowledge and domain knowledge for doing RA on network management. The specific domain knowledge along with the methodological knowledge forms a specific RA tool (Lin et al., 1994). For simplicity and readability, we set our focus on the RA method called *Class-object-diagramming-layer1*, which controls the identification of classes and objects for a given problem. Specifically, it takes the terms that describe the given problem as input and finds meaningful entities to produce a set of system classes and objects as output. This method accomplishes the work of the Class-object-diagramming-layer1 method specified in the Coad and Yourdon object-oriented analysis methodology (Coad & Yourdon, 1991). It needs the domain

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

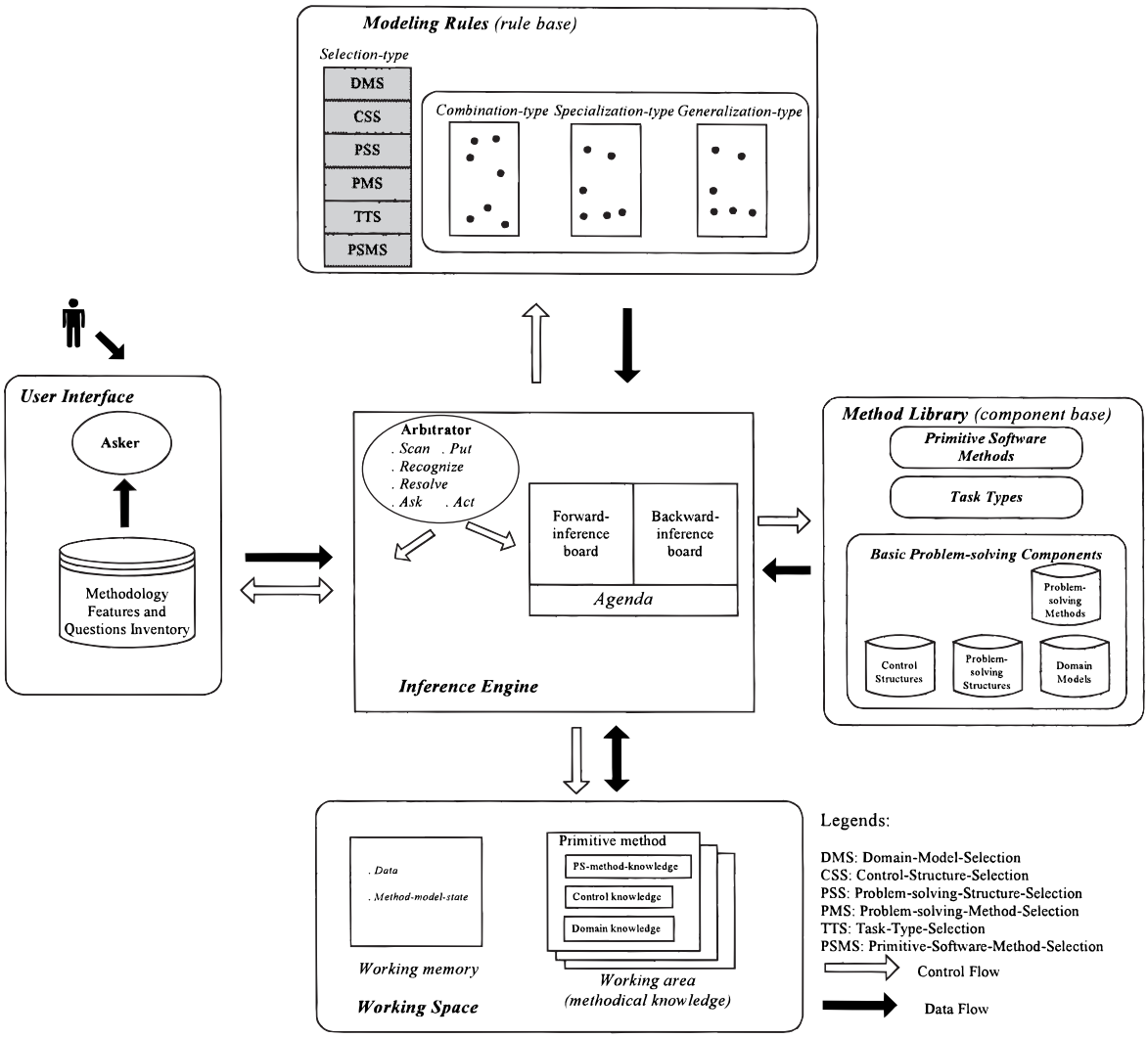


Fig. 12. Methodology modeling process.

knowledge that contains relevant information of domain objects to do RA. The generation of this RA method and the associated domain knowledge using the GAMP approach is illustrated in the following subsections.

4.1. Methodology-modeling process

First, we produce the methodological knowledge using the methodology modeling process. Figure 15 shows the desired methodological knowledge, that is, the *Class-object-diagramming-layer1* method. It is read as follows. The Has-task-knowledge slot uses *Design* as the first-level task type, which is further refined to *Parametric-design* subtype. The *Design* task type will guide the reasoning process to select *Proposing-checking-revising* as the problem-solving method (see the Has-PS-method-knowledge slot) and *Propose-criticize-modify* as the control structure (see the Has-control-knowledge slot). Both of them are refined according to the Parametric-design subtype. Proposing-checking-revising is refined to three constituent methods, that is, *Proposing*, *Checking*, and *Revising*, with Proposing further refined to *Find-object* and *Find-part* methods. Propose-criticize-modify structure is refined to *Domain-specific-match* structure for Proposing, *Feature-based-look-up* structure for Checking, and *Dependency-directed-backtracking* structure for Revising, respectively. According to the refinement of Proposing, Domain-specific-match is further refined to *Anchoring-search* corresponding to Find-object, and *Traveling-search* to Find-part. For supporting the operation of the control knowledge, the Has-domain-knowledge slot contains an *Anchoring* model for Anchoring-search, a *Traveling* model for Traveling-search, a *Constraining* model for Feature-based-look-up, and a *Fixing* model for Dependency-directed-backtracking.

In summary, the RA method uses the Find-object and Find-part problem-solving methods to propose all possible

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

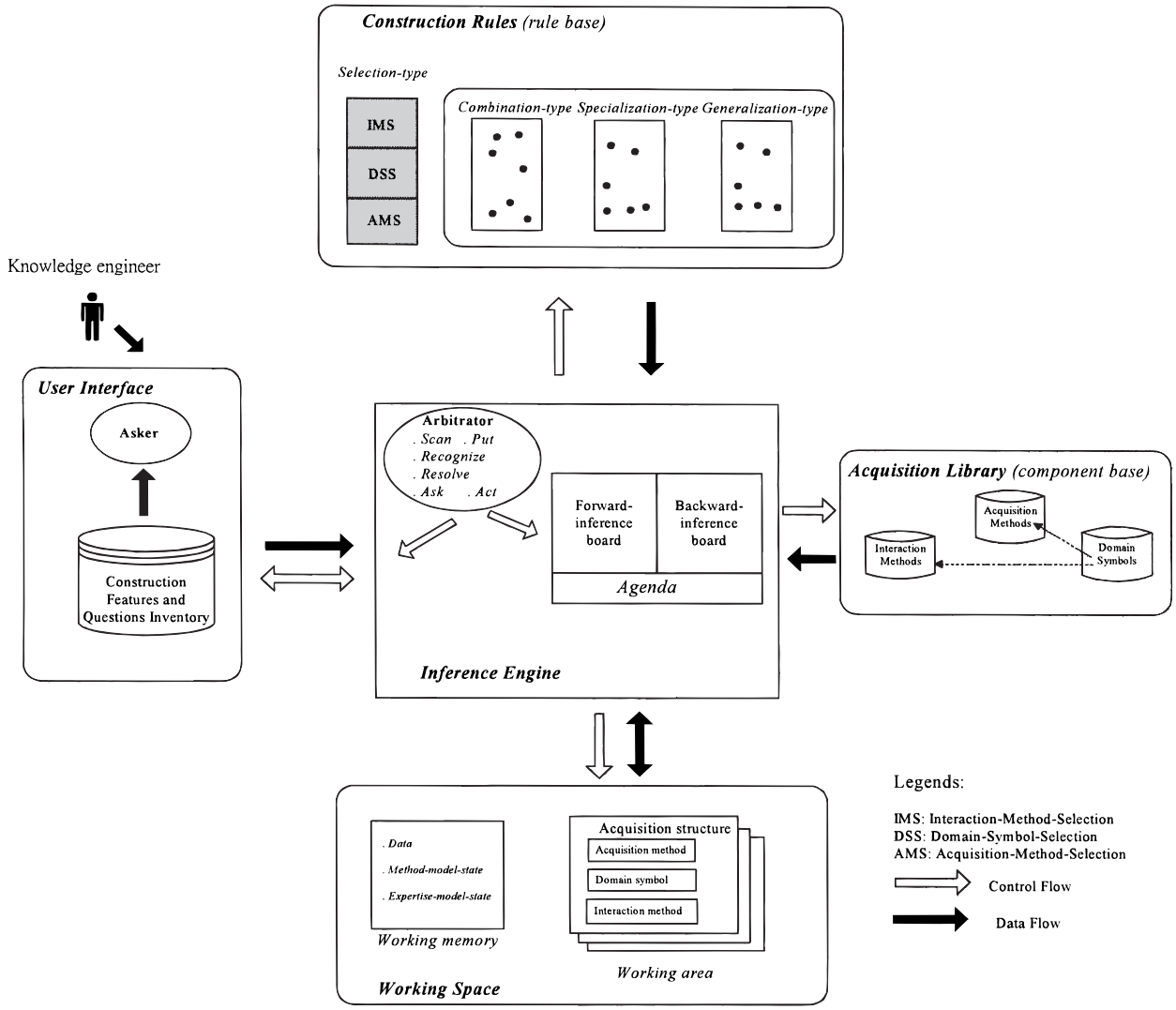


Fig. 13. Knowledge construction process.

objects/classes and their relationships for the given domain. It then uses Checking to check their reasonability using the constraint knowledge (represented by the Constraining model). It finally utilizes Revising to fix all constraint violations using the fix knowledge (represented by the Fixing model) to get the desired result. The detail execution logic of Find-object and Find-part is specified by Anchoring-search and Traveling-search, that of Checking is specified by Feature-based-look-up, and that of Revising by Dependency-directed-backtracking.

To create this RA method using the methodology modeling process, abstract methodology features that describe the properties of the problem need to be placed in the working memory first. In the network-management domain, the most important function is to monitor and control the network equipment. The RA process hence should focus on the definition, description, organization, and format of the data that moves across the interface between the system and its environment. Moreover, the stimuli and responses of the system are well structured and predefinable. Given this analysis, we place the following initial feature values in the working memory: "Monitoring/Control," "Messaging," "Data-oriented," and "Structured."

Now, the methodology modeling process starts by invoking the *Scan* operation to find probable modeling rules. In our example, the PSMS-type (Primitive Software Method Selection) rule set is the most probable. The *Put* operation is then invoked to place them on the backward-inference board. The *Recognize* operation follows to see how many of them are capable of being enabled. In our example, Primitive-software-method-selection#1, Primitive-software-method-selection#2, and Primitive-software-method-selection#5 rules constitute a conflict set (see Fig. 16). The *Resolve* operation chooses Primitive-software-method-selection#1 as the triggered rule according to the conflict-resolution strategy. With the help from the *Ask* operation to

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

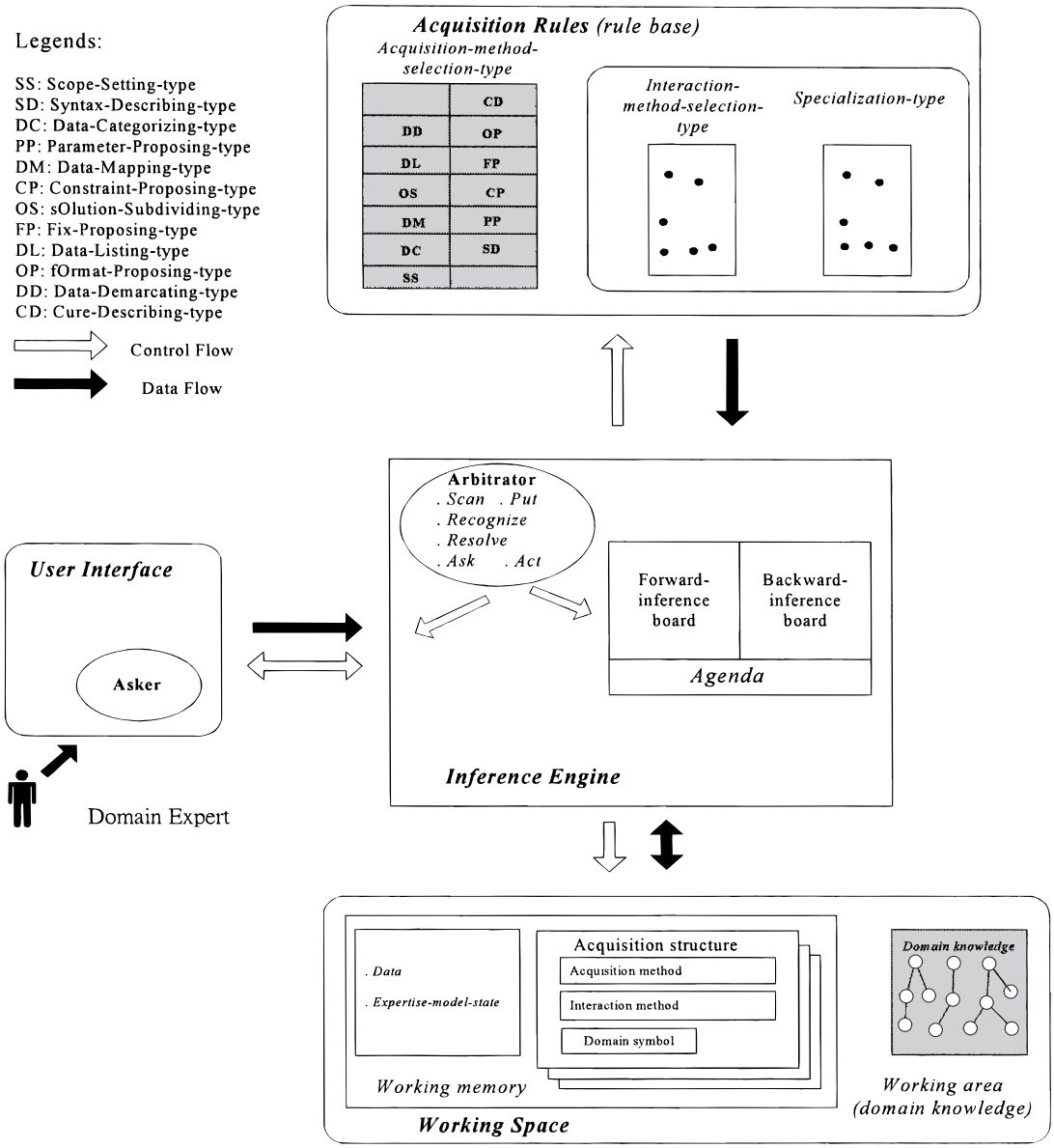


Fig. 14. Knowledge acquisition process.

get more values for the rule, the *Act* operation then fires it, which selects the *Entity-identification-classification* method as the primitive software method.

Following this similar line of reasoning, we can fire the Primitive-software-method-selection#2 rule to select the *Class-object-diagramming-layer1* method to refine the *Entity-identification-classification* method; fire the Task-type-selection#1 rule to select the *Design* task to fill the Has-task-knowledge slot; fire the Problem-solving-method-selection#1 rule to select the *Proposing-checking-revising* problem-solving method to elaborate the design task; and fire the Control-structure-selection#1 to select the *Propose-criticize-modify* control structure to specify the control. Some specialization-type rules are also fired to decompose the control structure into *Domain-specific-match*, *Feature-based-look-up*, and *Dependency-directed-backtracking* sub-control structures, which finally complete the slots of Figure 15.

4.2. Knowledge-construction process

Now, we produce the domain knowledge. Figure 17 shows the example domain knowledge. This section talks about the knowledge-construction process. Figure 18 shows an example acquisition structure to be produced by the process for the "Equipment" node in Figure 17.

The knowledge construction process can start only after the working space contains method-model-state. Follow-

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

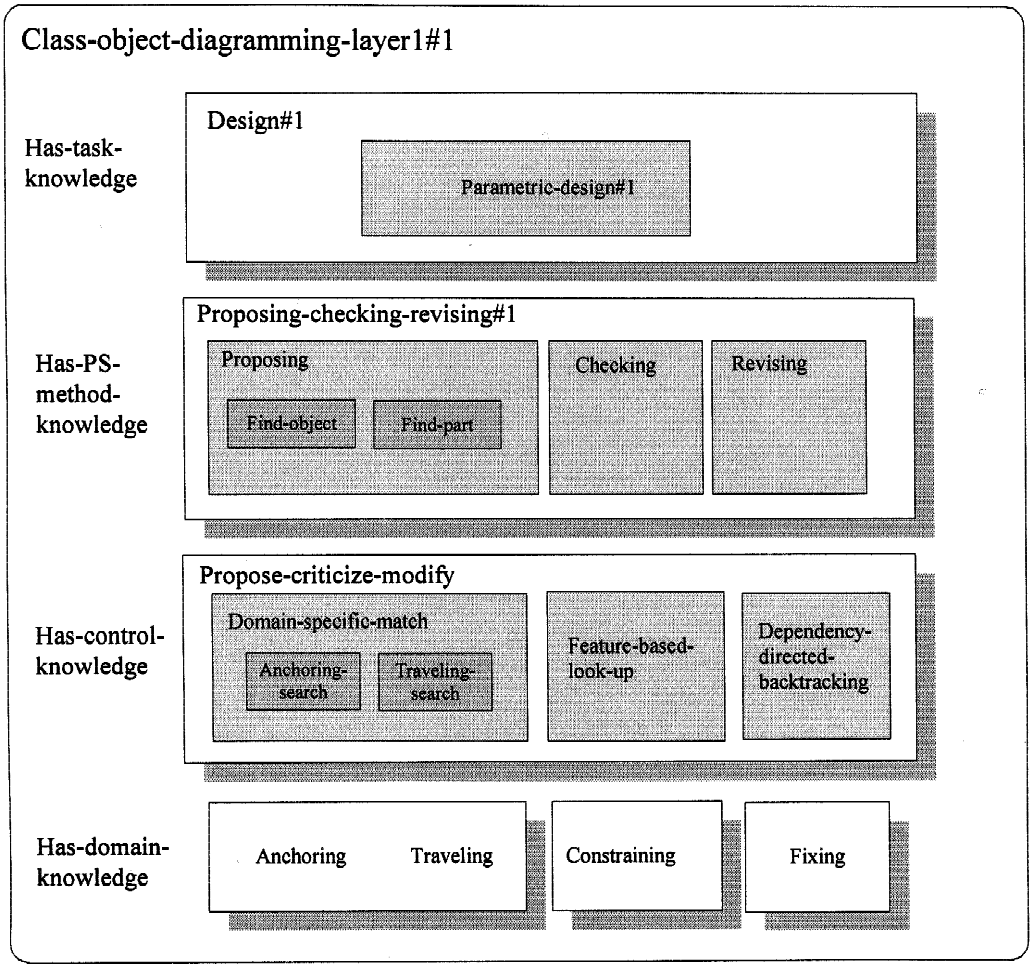


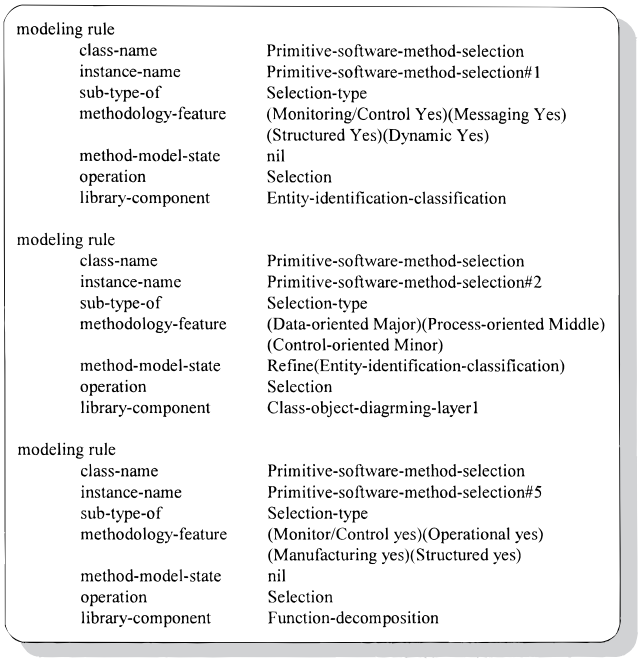
Fig. 15. The Class-object-diagramming-layer1#1 primitive method.

ing the same example, the specific methodological knowledge now is Class-object-diagramming-layer1, which has been elaborated into a Parametric-design task with three corresponding problem-solving methods—Proposing, Checking, and Revising. Accordingly, we have the following values for the method-model-state (in the working memory): "Problem-solving-method Proposing," "Problem-solving-method Checking," and "Problem-solving-method Revising." In addition, similar to the methodology modeling process, the knowledge construction process needs some basic construction features that describe the properties of the domain. The most significant things in this network management domain are the functions of network management, the agents that execute these functions, and the equipment managed by the agents. For simplicity, we only demonstrate the acquisition of knowledge for equipment. With this analysis, we can place the following values in the working memory as the domain's major features: "Object Yes" and "Taxonomic Yes."

Now, the knowledge-construction process starts by invoking the *Scan* operation to find probable construction rules. In the example, the AMS-type (Acquisition Method Selection) rule set is the most probable. The *Put* operation is then invoked to place them on the backward-inference board. The *Recognize* operation follows to produce a conflict set. In our example, Acquisition-method-selection#1, Acquisition-method-selection#2, and Acquisition-method-selection#3 rules form a conflict set (Fig. 19). The *Resolve* chooses Acquisition-method-section#1 to be triggered. Because all feature values of the rule are matched, *Act* immediately fires the rule and produces *Parameter-proposing* as the basic acquisition method.

Following this line of reasoning, we can fire the Domain-symbol-selection#1 rule to select the *Parameter-type* domain symbol for the Parameter-proposing acquisition method; fire the Instantiation#1 rule to get relevant equipment and their corresponding attributes to instantiate the Parameter-type domain symbol; and fire the Interaction-method-selection#1 rule to select the *Table-driven* interaction method for the instantiated domain symbol. This eventually leads to the partially completed acquisition structure for the "Equipment" node as shown in Figure 18. After the knowledge construction process, each of the nodes in Figure 17 will contain a similar acquisition structure.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press



modeling rule	
class-name	Primitive-software-method-selection
instance-name	Primitive-software-method-selection#1
sub-type-of	Selection-type
methodology-feature	(Monitoring/Control Yes)(Messaging Yes) (Structured Yes)(Dynamic Yes)
method-model-state	nil
operation	Selection
library-component	Entity-identification-classification
modeling rule	
class-name	Primitive-software-method-selection
instance-name	Primitive-software-method-selection#2
sub-type-of	Selection-type
methodology-feature	(Data-oriented Major)(Process-oriented Middle) (Control-oriented Minor)
method-model-state	Refine(Entity-identification-classification)
operation	Selection
library-component	Class-object-diagrming-layer1
modeling rule	
class-name	Primitive-software-method-selection
instance-name	Primitive-software-method-selection#5
sub-type-of	Selection-type
methodology-feature	(Monitor/Control yes)(Operational yes) (Manufacturing yes)(Structured yes)
method-model-state	nil
operation	Selection
library-component	Function-decomposition

Fig. 16. Sample modeling rules.

4.3. Knowledge-acquisition process

Now we can complete the generation of domain knowledge by invoking the knowledge-acquisition process, that is, call the acquisition rules to do actual domain-knowledge acquisition. Similar to the previous processes, this process starts by invoking the *Scan* operation to find probable acquisition rules. In the example, the PP-type rule set contains the most probable rules. The *Put* operation is then invoked to place them on the backward-inference board. Because the set only contains the Parameter-proposing-type-selection#1 rule, it is selected by the *Resolve* operations to be triggered. Finally, the *Act* operation fires it to select all the acquisition methods of type "Parameter-proposing." Note that the domain symbol contained in this acquisition method is "Equipment," which will be used later to select proper interaction methods.

Continuing with this reasoning, we will finally fire the Domain-symbol-type-interaction#1 rule to select an interaction method for "Equipment"; and fire the instantiation#1 rule to use the selected interaction method to ask the user for the relevant values for "Equipment." In this example, the user is requested to key in "ADM600" for instance-name and its description—"An add–drop multiplexer for SONET network." This process continues until all the acquisition structures associated with the nodes in Figure 17 are completely materialized, which thus completes the domain-knowledge-acquisition process.

5. DISCUSSION AND CONCLUSION

We have described GAMP as a generic-ontology-based platform for RA-tool generation. GAMP allows software engineers to derive domain-appropriate RA-tools for different, enterprise-wide, or heterogeneous applications. It advocates the concept of a software methodology ontology that contains methodology features, modeling rules, and a method library. The modeling rules work as the basic procedures to furnish skeletal primitive software methods in the method library and produce domain-related methodological knowledge. GAMP also uses a knowledge-acquisition ontology for generating domain knowledge. This is done by using the construction rules to construct domain-specific acquisition structures from an acquisition library, and by using the acquisition rules to follow these acquisition structures to acquire domain knowledge. Coupling the methodological knowledge with the domain knowledge forms an RA tool.

We have demonstrated how GAMP is used to derive an RA tool that assists the end user to acquire and formalize a requirement specification for the network management domain. As a matter of fact, to verify that this approach is generic in the sense that it works in different domains, Chung-Hwa Telecommunications has applied it in other problems, to name a few, an Internet management system, HiView (Lin et al., 1997), and a customer-service system, TOPS (Liu & Kuo, 1998). The major function of HiView is to monitor and control the software and hardware of Hi-Net, an Internet Service Provider in Taiwan, so that the network can work properly. The software to be monitored includes terminal servers, domain name servers, world-wide-web servers, proxy servers, mail servers, etc. The hardware includes routers, hubs, gateways, switchings, etc. This domain is similar to SONET. Application of GAMP in the domain essentially can follow the same process. First, the generated-methodological knowledge can be directly reused in the domain. Second, the same acquisition structure is generated for the domain except that during the knowledge construction process the user is requested to enter Internet-specific domain symbols, such as domain name servers, IP addresses, socket ports, etc. This makes the acquisition structures Internet-specific and facilitates the subsequent generation of Internet-specific domain knowledge. Coupling the Internet-specific domain knowledge with the same methodological knowledge, we thus generate an RA tool for HiView.

TOPS (Telecom Operation Service System) provides Chung-Hwa Telecommunications with a one-stop-shopping service for home subscribers, enterprise customers, and business salesmen. Its services include provisioning of telephones or digital private lines, billing processing, and planning of PBXs, etc. It affects most subscribers of the company, that is about 10 million people. The company's efficient operation, therefore, is highly dependent upon its performance. The basic key requirements for TOPS are as follows: One is to provide a *flow control* that can customize an order flow according to its specification. Another is to properly manage a *unified*, *common database* that ensures all the data are modeled precisely and consistently. Still another is to provide a *highly interactive interface* for easy access of the system by the customer.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

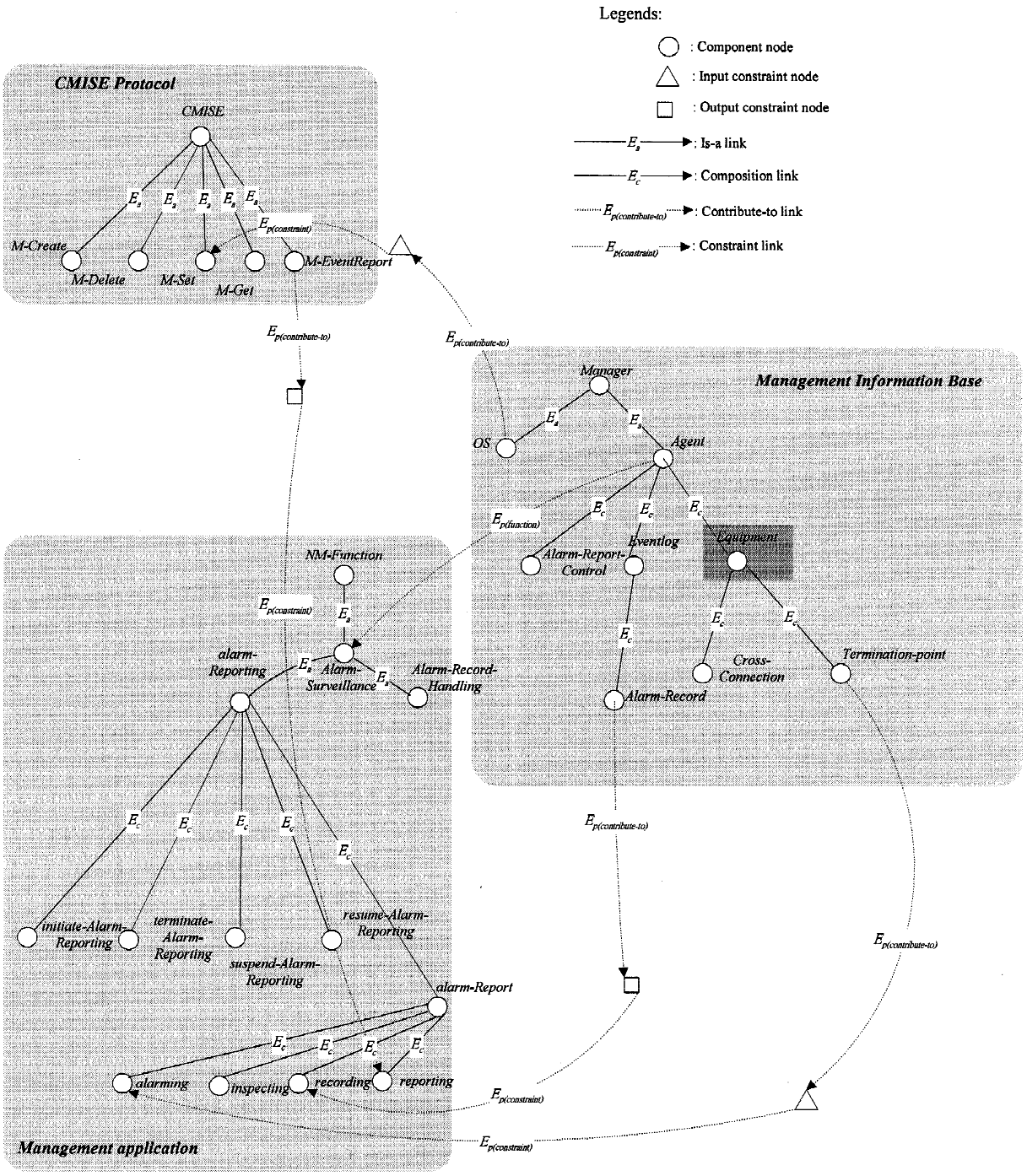


Fig. 17. Diagrammatic representation for the domain knowledge of SONET ADM600.

This application is quite a different domain and much more complex than SONET. The generation of its RA tool can be done by GAMP too. First, the user analyzes the above requirements to get a set of applicable abstract methodology features, as underlined in Figure 5. Then, following the methodology-modeling process, these values are entered to the system to invoke the modeling rules, which finally generate the software methods listed below. A *Dataflow-diagramming* method, a *Process-dependency-diagramming* method, and an *Entity-dataflow-diagramming* method are generated corresponding to the features "Operational," "Process-oriented," "Inventory-management," "Structured," and "Transaction-processing" that meet the requirement of *Flow control*. An *Entity-relationship-diagramming* method, a *Data dictionary* method, and a *Domain-charting* method are generated corresponding to thefeatures "Stable," "Physical-system," "Static," "Structured," "Inventory-management," and "Data-oriented" that

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

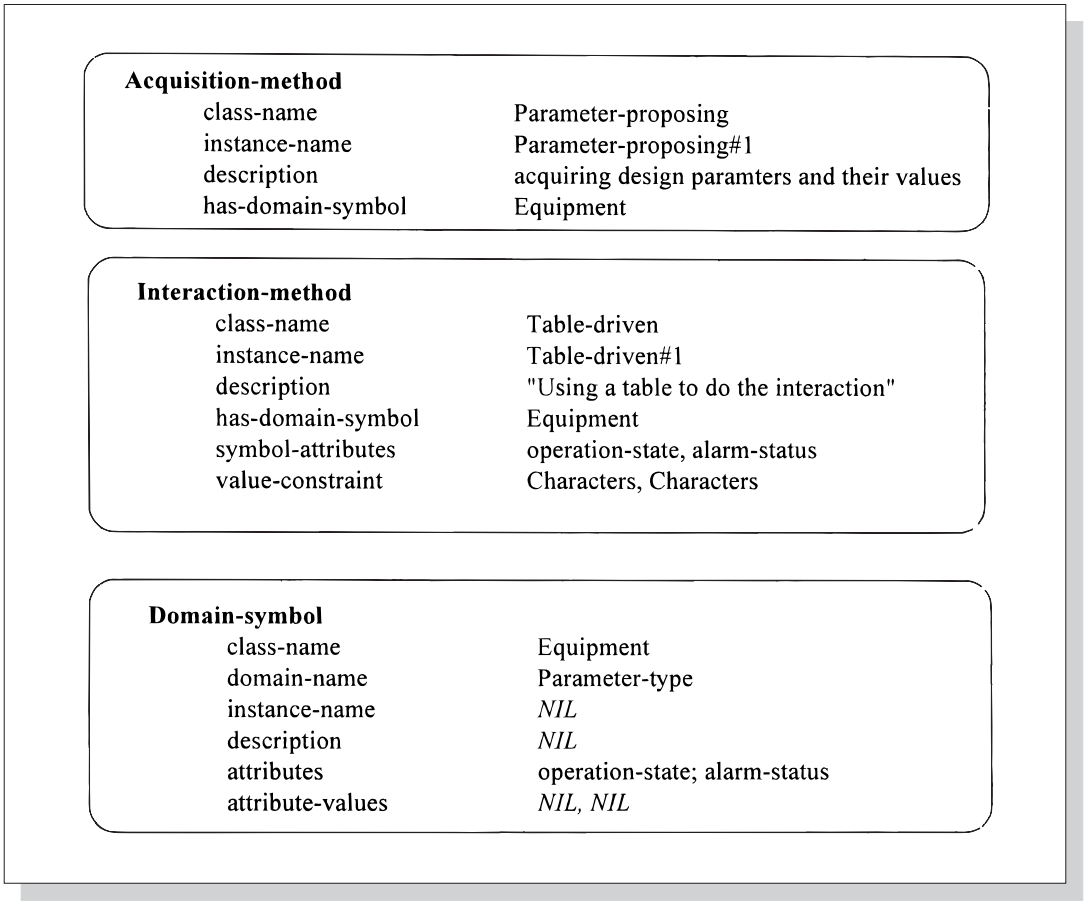


Fig. 18. Acquisition structure for "Equipment."

meet the requirement of *unified, common database*. Finally, an *Object-communication-model* method and an *Object-access-model* method are generated corresponding to the features "Control-oriented," "Unstructured," "Messaging," "Marketing," and "Slow-evolving" that meet the requirement of *highly interactive interface*. These software methods will then guide the construction rules of GAMP to construct TOPS-specific acquisition structures to produce domain knowledge. Coupling both then forms an RA tool for TOPS.

The advantages of using GAMP to generate specific RA tools are summarized below:

1. GAMP is ontology based. It analyzes, abstracts, and integrates existing software-development methodologies and knowledge-acquisition techniques into proper ontologies. It supports the integration, sharing, and reuse of software methodologies and knowledge acquisition techniques, which alleviates the problems associated with the correct generation of requirement specification.
2. GAMP is generic. The generation of RA tools for different domains are properly supported by the ontologies.
3. The generated RA tools from GAMP contain methodological knowledge and domain knowledge, either of which is readily reused. For example, we can replace the methodological knowledge by another RA method to generate a new RA tool. Or, vice versa, we can replace the domain knowledge with a new one for a different domain. This improves system flexibility and productivity.
4. The generation process of RA tools using GAMP is conducted by various rules, which helps the development of correct RA tools and in turn helps the production of correct requirement specification. Furthermore, the rules can be modified to meet different considerations, which further increases the flexibility of the approach.

Currently, the performance of the generated RA tools is sensitive to the user. Although he only needs to know how to use GAMP without going into details of all the ontologies, he needs to be familiar with the problem domain to properly transfer the meanings of the requirement into the features values for GAMP to generate RA tools. We are working on hybridizing *case-based reasoning* with other techniques to alleviate this limitation (Hsu & Ho, 1998). We are

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

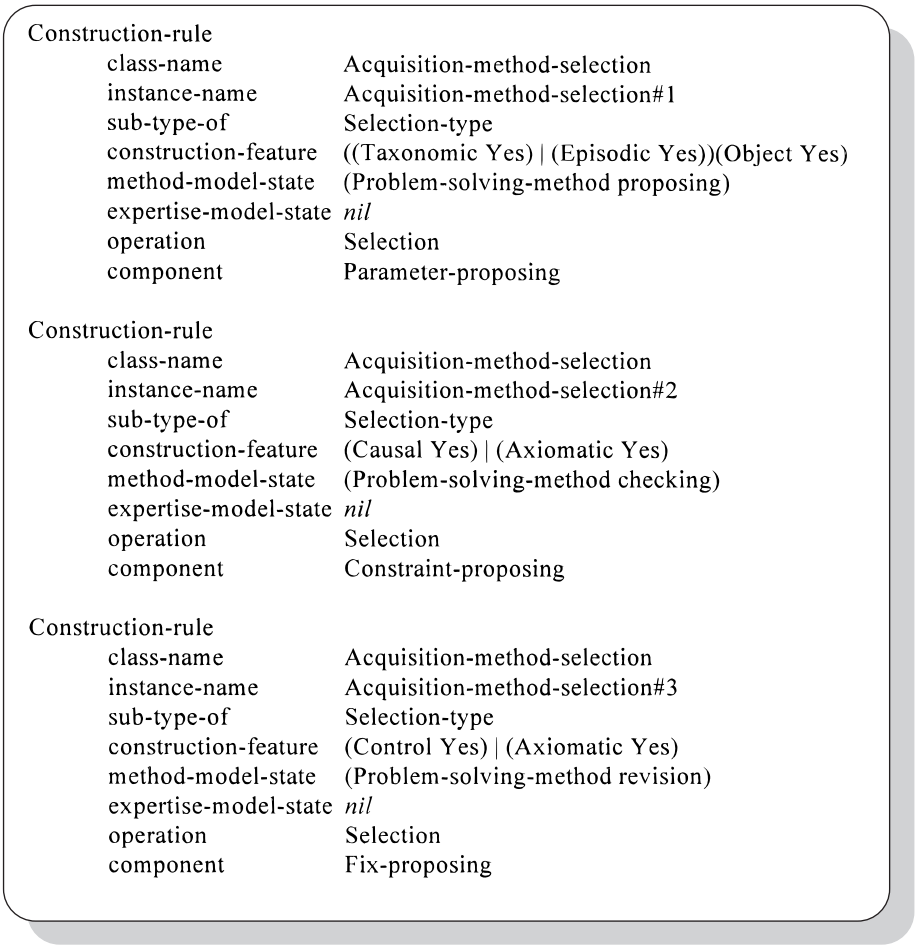


Fig. 19. Three sample construction rules.

also working on developing an ontology of software-design methods for GAMP so that it can generate a knowledge-based system to create software design, which will be used to translate the requirement specification produced by the current RA tool into a design specification.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments in revising the paper. Thanks also go to Mr. Shyang-Shi Chen and Mr. Win-Pin Kang for their assistance in system implementation. This work was supported in part by the National Science Council, Taiwan, under grants NSC81-0408-E-011-563 and NSC82-0408-E-011-183, and by the Telecommunication Laboratories of Chung-Hwa Telecommunications, Taiwan, under grant TL-83-3204.

REFERENCES

Alberts, L.K. (1993). *YMIR: An ontology for engineering design*. PhD thesis, University of Twente, Netherlands.

ANSI T1M1.5/90-010 (1990). *Generic network model for interface between operation systems and network elements*. American National Standards Institute, New York.

ANSI T1.105 (1995). *Synchronous optical network (SONET)—basic description including multiplex structure, rates, and formats*. American National Standards Institute, New York.

Bennett, J.S. (1985). ROGET: A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system. *Journal of Automated Reasoning 1*, 49–74.

Bickerton, M.J., & Siddiqi, J. (1993). The classification of requirements engineering methods. *Proc. IEEE Symposium on Requirements Engineering*, San Diego, CA, 182–185.

Binder, R.V., & Tsai, J.J.P. (1992). KB/RMS: An intelligent assistant for requirement definition. *International Journal of Artificial Intelligence Tools 1(4)*, 503–522.

Black, U., & Waters, S. (1996). *SONET & T1: Architectures for digital transport networks*. Prentice-Hall Inc., Englewood Cliffs, NJ.

Boose, J.H. (1984). Personal construct theory and the transfer of human expertise. *Proc. The National Conference on Artificial Intelligence (AAAI-84)*, Austin, Texas, 27–33.

Boose, J.H. (1989). A survey of knowledge acquisition techniques and tools. *Knowledge Acquisition 1*, 3–37.

Boose, J.H. (1990). Uses of repertory grid-centered knowledge acquisition tools for knowledge-based systems. In *The Foundations of Knowledge Acquisition*, (Boose, J.H, & Gaines, B.R., Eds.), pp. 61–84. Academic Press, London.

Boose, J.H., & Gaines, B.R. (1988). *Knowledge acquisition tools for expert systems*. Academic Press, San Diego, CA.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

Boose, J.H., & Gaines, B.R. (1990). *The foundation of knowledge acquisition*. Academic Press, San Diego, CA.
Brownston, L., Farrell, R., Kant, E., & Martin, N. (1985). *Programming expert systems in OPS5: An introduction to rule-based programming*. Addison-Wesley, Reading, MA.
Budd, M., & Woods, E. (1998). *Ovum evaluates: CASE products*. Ovum Ltd., London, UK.
Chander, P.G., Shinghal, R., & Radhakrishnan, T. (1997). Using goals to design and verify rule bases. *Decision Support Systems 21(4)*, 281–305.
Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert 1(3)*, 23–30.
Chandrasekaran, B., Johnson, T.R., & Smith, J.W. (1992). Task-structure analysis for knowledge modeling. *Communications of the ACM 35(9)*, 124–137.
Chandrasekaran, B., Josephson, J.R., & Benjamins V.R. (1998). The ontology of tasks and methods. *Proc. Workshop on Knowledge Acquisition, Modeling and Management*, Banff, Canada.
Chien, C.C., & Ho, C.S. (1994). A primitive-based generic approach to knowledge acquisition. *Knowledge Acquisition 6*, 215–242.
Chu, B.H. (1995). *A graph-theoretical method for verifying, validating and refining constraint-based knowledge bases*. Master Thesis, Department of Electronic Engineering, National Taiwan University of Science and Technology, Taiwan, China.
Coad, P., & Yourdon, E. (1991). *Object-oriented analysis*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
Czuchry, A.J. (1988). KBRA: A new paradigm for requirements engineering. *IEEE Expert 3(4)*, 21–35.
David, E.M., & Gretchen, I.P. (1992). A research typology for object-oriented analysis and design. *Communications of the ACM 35(9)*, 35–47.
Davis, A.M. (1993). *Software requirements: Objects, functions, and states*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
Destor, C., Scortesse, A., & De Man, J. (1990). Object-oriented techniques for switching systems. *Electrical Communication 64(4)*, 371–379.
Dion, R. (1993). Process improvement and the corporate balance sheet. *IEEE Software 10(4)*, 28–35.
Eshelman, L. (1988). MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In *Automating Knowledge Acquisition for Expert Systems*, (Marcus, S., Eds.), pp. 37–80. Kluwer Academic, Norwell.
Fichman, R.G., & Kemerer, C.F. (1992). Object-oriented and conventional analysis and design methodologies: Comparison and critique. *IEEE Computer 25(10)*, 22–39.
G.784 (1994). *Synchronous digital hierarchy (SDH) management*. ITU-T Recommendation, Geneva, Switzerland.
Gaines, B.R., & Boose, J.H. (1988). *Knowledge acquisition for knowledge-based systems*. Academic Press, San Diego, CA.
Gomaa, H., Kerschberg, L., Sugumaran, V., Bosch, C., Tavakoli, I., & O'Hara, L. (1996). A knowledge-based software engineering environment for reusable software requirements and architectures. *Automated Software Engineering 3(3)*, 285–307.
Green, C., Luckham, D., Balzer, R., Cheatham, T., & Rich, C. (1986). Report on a knowledge-based software assistant. In *Readings in Artificial Intelligence and Software Engineering*, (Rich, C., & Waters, R.C., Eds.), pp. 377–428. Morgan Kaufmann, San Mateo, CA.
Gruber, T.R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition 5*, 199–220.
Gruber, T.R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human and Computer Studies 43(5/6)*, 907–928.
Guarino, N. (1995). Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human and Computer Studies 43(5/6)*, 625–640.
Guarino, N. (1997). Understanding, building and using ontologies. *International Journal of Human and Computer Studies 46(2/3)*, 293–310
Guarino, N. (1998). Formal ontology and information systems. *Proc. Formal Ontology in Information Systems (FOIS'98)*, Trento, Italy.
Guenther, W., & Wackerbarth, G. (1993). Object-oriented design of ISDN call-processing software. *IEEE Communications 31(4)*, 40–45.
Hsu, C.C., & Ho, C.S. (1998). An intelligent hybrid system for knowledge acquisition. *Proc. 11th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, Castellon, Spain.
ISO 7498-4 (1989). *Information processing systems—open systems interconnection—basic reference model—part 4: Management framework*. International Organization for Standardization, Geneva, Switzerland.
ISO 9595 (1990). *Information processing systems—open systems interconnection—common management information service definition*. International Organization for Standardization, Geneva, Switzerland.
ISO 9000-3 (1991). *Guidelines for the application of ISO 9001 to the development, supply, and maintenance of software*. International Organization for Standardization, Geneva, Switzerland.
Johnson, W.L., Benner, K.M., & Harris, D.R. (1993). Developing formal specifications from informal requirements. *IEEE Expert 8(4)*, 82-90.
Johnson, W.L., & Feather, M.S. (1991). Using evolution transformations to construct specifications. In *Automating Software Design*, (Lawry, M.R., & McCartney, R.D., Eds.), pp. 65–92. AAAI Press, Menlo Park, CA.
Johnson, W.L., Feather, M.S., & Harris, D.R. (1992). Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering 18(10)*, 853–869.
Jorgensen, P. (1986). Minireview: Software requirement specification. *Proc. IEEE 10th Ann. Int'l Computer Software and Applications Conf. (COMPSAC86)*, p. 182.
Kahn, G., Nowlan, S., & McDermott, J. (1985). MORE: An intelligent knowledge acquisition tool. *Proc. 9th Int. Joint Conf. Artificial Intelligence (IJCAI-85)*, 581–584.
Kawaguchi, A., Motoda, H., & Mizoguchi, R. (1991). Interview-based knowledge acquisition using dynamic analysis. *IEEE Expert 6(10)*, 47–60.
Kiper, J.D. (1992). Structural testing of rule-based systems. *ACM Trans. Software Engineering and Methodology 1(2)*, 168–187.
Kitto, C.M., & Boose, J.H. (1990). Selecting knowledge acquisition tools and strategies based on application characteristics. In *The Foundations of Knowledge Acquisition*, (Boose, J.H., & Gaines, B.R., Eds.), pp. 257–268. Academic Press, London.
Klerer, S.M. (1988). The OSI management architecture: An overview. *IEEE Network 2(2)*, 20–29.
Laird, J., Newell, A., & Rosenbloom, P. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence 33*, 1–64
Lee, C.H., Sun, Y.S., Tarng, M.J., & Kao, Y.Y. (1997). The design of operation management software for ADM150 equipment. *Telecommunication Labs. Technical Journal 27(4)*, 1–21.
Lenat, D.B., & Guha, R.V. (1990). *Building large knowledge-based systems: Representation and inference in the CYC project*. Addison-Wesley, Reading, MA.
Lin, C.Y.I., & Ho, C.S. (1996). *A primitives-based meta-platform for requirement analysis*. Report No. TR85-04. Department of Electronic Engineering, National Taiwan University of Science and Technology, Taiwan, China.
Lin, C.Y.I., & Ho, C.S. (1998*a*). Generating domain-specific methodical knowledge for requirement analysis based on methodology ontology. Accepted for publication in *Information Sciences—An International Journal*.
Lin, C.Y.I., & Ho, C.S. (1998*b*). *Knowledge representation, reasoning, and evaluation for requirement analysis: the GAMP approach*. Technical Report, Department of Electronic Engineering, National Taiwan University of Science and Technology, Taiwan, China.
Lin, C.Y.I., Ho, C.S., & Lee, H.M. (1994). NMRA: A knowledge-based system for requirement analysis of network management software. *Proc. 3nd Pacific Rim Int. Conf. Artificial Intelligence (PRICAI-94)*, 1000–1005.
Lin, C.Y.I., Jou, T.L., & Leu, C.Y. (1991). Design of OSI network management in SONET ADM600. *Proc. International Symposium on Communications*, 716–720.
Lin, L.S., Chen, M.L., & Lu, C.S. (1997). HiView: A network management system for HiNet. *Proc. Telecommunication Technology Symposium*, 162–170.
Liu, H.Y., & Kuo, C.C. (1998). *The design of a service order processing system*. Report No. TR87-010. Telecommunication Laboratories, Chung-Hwa Telecommunications, Chungli, Taiwan, China.
Lu, F.S., Lin, C.Y.I., & Liang, S.W. (1994). The design of an exceptional case expert system for SOPS. *Telecoms Technical Quarterly 13(4)*, 335–345.
Maiden, N.A.M., & Rugg, G. (1996). ACRE: Selecting methods for re-

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press

quirements acquisition. *Software Engineering Journal 11(3)*, 183–192.
Marcus, S. (1988). *Automating knowledge acquisition for expert systems*. Kluwer Academic, Norwell.
Marcus, S. (1990). *Knowledge acquisition: Selected research and commentary*. Kluwer Academic, Norwell.
Marcus, S., & McDermott, J. (1989). SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence 39(1)*, 1–37.
McGraw, K.L., & Harbison-Briggs, K. (1989). *Knowledge acquisition principles and guidelines*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
Mizoguchi, R., Tuerino, Y., & Ikeda, M. (1995). Task analysis interview based on task ontology. *Expert Systems with Applications 9(1)*, 15–25.
Ouyang, R.L. (1998). *A study on multi-strategy learning of operation knowledge for the steam generator tube ruptured accidents of nuclear power plants*. Master's thesis. Department of Electronic Engineering, National Taiwan University of Science and Technology, Taiwan, China.
Paulk, M. (1993). Capability maturity model, version 1.1. *IEEE Software 10(4)*, 18–27.
Reubenstein, H.B., & Waters, R.C. (1991). The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering 17(3)*, 226–240.
Rosenbloom, P.S., Laird, J.E., & Newell, A. (1993). *The soar papers: Readings on integrated intelligence*. MIT Press, Cambridge, MA.
Schreiber A.T., & Wielinga, B.J. (1993). Comparing KADS to conventional software engineering. In *KADS: A Principled Approach to Knowledge-Based System Development*, (Schreiber, A.T., Wielinga, B.J., & Breuker, J.A., Eds.), pp. 151–165. Academic Press, London.
Schreiber, A.T., Wielinga, B.J., & Hoog, R.D. (1994). CommonKADS: A comprehensive methodology for KBS development. *IEEE Expert 9(6)*, 28–37.
Schreiber, G., Wielinga, B., & Jansweijer, W. (1995). The KAKTUS view on the 'O' word. *Proc. 14th Int. Joint Conf. Artificial Intelligence (IJCAI95) Workshop on Basic Ontological Issues in Knowledge Sharing*.
Short, W. (1991). Methodology integration: Evolution of information engineering. *Information and Software Technology 33(9)*, 720–731.
Standish Group (1994). *CHAOS*. The Standish Group International Inc., Dennis, MA.
Tijerino, Y.A., Ikeda, M., Kitahashi, T., & Mizoguchi, R. (1993). A methodology for building expert systems based on task ontology and reuse of knowledge—underlying philosophy of a task analysis interview system MULTIS. *Journal of Japanese Society for Artificial Intelligence 8(4)*, 476–487.
Wang, J.T., Lee, Y.S., & Chen, H.S. (1989). The planning and development of a service order processing system. *Telecoms Technical Quarterly 9(2)*, 1–28.
Wielinga, B.J., & Schreiber, A.T. (1993). Reusable and sharable knowledge bases: A european perspective. *Proc. Int. Conf. Building and Sharing of Very Large-Scaled Knowledge Bases*, 103–115.
van Heijst, G., Schreiber, A.T., & Wielinga, B.J. (1997). Using explicit ontologies in KBS development. *International Journal of Human and Computer Studies 46(2/3)*,183–292.

Chau-Young Ivan Lin is the Head of Information Group in the Program Coordination Department, Telecommunication Laboratories, Chung-Hwa Telecommunications Co. Ltd. His current research interests include knowledge-based software engineering, object-oriented software methodology and technology, and knowledge-based systems. He received the B.S. and M.S. degrees from the FengChia University in 1981 and 1987, respectively. He is currently working toward his Ph.D. degree in the Department of Electronic Engineering, National Taiwan University of Science and Technology. Mr. Lin is a member of the Taiwanese Association for Artificial Intelligence, OSI ODA SIG Taiwan, and the IEEE Computer Society.

Cheng-Seen Ho is a Professor at the Department of Electronic Engineering, National National Taiwan University of Science and Technology. He worked with the Taiwan Power Company, Long Distance Telecommunication Administration, and Telecommunication Laboratories before joining the department. His current research interests include variable Quality-of-Service systems, intelligent agents, hybrid systems, AI in software engineering, and AI in autonomous control systems. He received the B.S., M.S., and Ph.D. degrees from the Department of Electrical Engineering, National Taiwan University in 1976, 1982, and 1987, respectively. Dr. Ho is a member of AAAI, ACM, IEEE Computer Society, Institute of Information and Computing Machinery, International Society of Applied Intelligence, and Taiwanese Association for Artificial Intelligence (TAAI). He served as the first-term president of TAAI during 1995–1996.

https://doi.org/10.1017/S0890060499131044 Published online by Cambridge University Press