TR-87-014

A DEVELOPMENT METHOLODOGY FOR OFFICE INFORMATION SYSTEMS BASED ON THE SOCIETY MODEL

TR-87-014

A DEVELOPMENT METHOLODOGY FOR OFFICE INFORMATION SYSTEMS BASED ON THE SOCIETY MODEL

主 持 人：郭德成

研究助理：洪炯宗

陈寶華

中央研究院資訊科學研究所

INSTITUTE OF INFORMATION SCIENCE

ACADEMIA SINICA

中華民國 七十六 年九月

Contents

Abstract

A general framework aiming at the development of a society model based office information system is developed. It includes a modeling tool and associated developing environment. The modeling tool is based on the society model which advocates the concept of knowledge-based office agents for constructing an office system so that the system can acquire sufficient knowledge from its environment and successfully cope with real world problems. To improve the knowledge engineering process, knowledge tables are used as a unified knowledge representation method to accommodate knowledge of various types of office agents into a single framework. A systematic, society model based methodology is then proposed, which is tailored for the development of knowledge-based office systems. It features a knowledge-based supporting environment to help manage the whole system development process and to facilitate the rapid prototyping of a knowledge-based office system.

1. Introduction

We have developed a society model in [HO86, KUO85] for office systems, which can accommodate various aspects of knowledge of a real-world office. A knowledge representation structure, knowledge tables, which can represent various types of knowledge in a unified structure has been developed in [CHAN86a]. This report will combine these concepts together and develop a methodology which includes an automated supporting environment for the development of a knowledge-based office system.

Section 2 starts with a survey of current software development models. Based on the study, along with the requirement of a knowledge-based office system, we then develop a society model based methodology for the development of an office information system in Section 3. Section 4 designs a component of the knowledge-based support system in the methodology called Office Automation Consultant as a knowledge table system. Finally developing a whole office system through the Office Automation Consultant is summarized in Section 5.

2. A Survey of Software Development Models

The main purpose of a software development model is to provide a framework for problem solving within which various methodologies and techniques can be applied to improve the

quality, efficiency, productivity, manageability, maintainability, and evolutionary capability of the developed software system. Several models associated with a variety of techniques have been proposed. Each of them has its own feature. We will investigate four of them ranging from the conventional human-oriented model to the knowledge-based computer-oriented model [RAMA87]. This observation will serve as a foundation for the development of an appropriate methodology for knowledge-based office systems.

2.1 Conventional Models

The most salient feature of the conventional software development model is that it is geared to program development and project management by humans. This human-oriented, management-directed feature makes early stages of the development process focus on the specification of external behavior of the problem domain as well as the design of overall intermodule mechanisms. Voluminous documentation might be produced during these phases. If no common understanding is enforced on these documents, any misinterpretation might seed errors into the design process.

The maintenance and enhancement of a software system developed under the model is performed on the implemented program. Usually, it makes the maintenance stage a cost

intensive phase in the whole development process. A simplified general structure of the conventional model is illustrated in Figure 1.

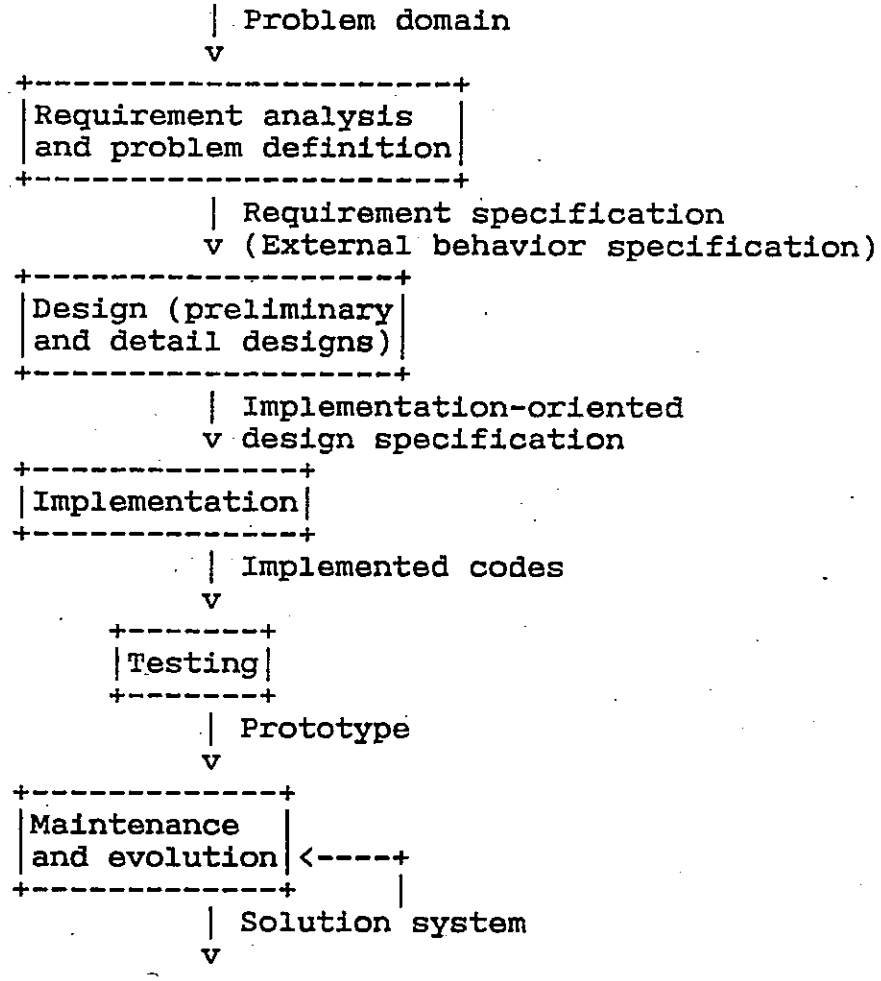

Fig. 1 Conventional Model

Five stages are incorporated in the conventional model. Each stage has documentary output that serves as the input to the next stage. Basically, early stages specify an informal behavior abstraction of what is computed. This abstrac-

tion is progressively refined in the later stages into a formal implementation of how the behavior can be realized.

Specifically, the model begins with the requirement analysis and problem specification. Ideally, both nonfunctional requirements and functional requirements have to be identified and specified. Nonfunctional requirements refers to various constraints imposed by users, for instance, system performance, reliability, quality, security, system working environment, maintenance contract, cost constraint, etc. (see [YEH84] for a more complete list). Functional requirements specify what the target system does. Both of the requirements were specified informally, e.g., using natural languages, during the early years of software engineering partly because of the lack of suitable technology. Although natural languages are easy for humans to understand, they are not rigid enough to eliminate any ambiguity. A variety of specification techniques and/or automated tools with predefined common terminology for the specification of problem domains have been proposed trying to "formalize" the specification and reduce ambiguity. Most of them are aiming at functional requirements due to no comprehensive theory or methodology for nonfunctional requirements specification. For instance, HIPO advocates a function-oriented decomposition technique to elicit functional requirements from users

[JONE76]. SADT provides a decomposable dataflow method as well as a specification language SA for system behavior specifications [ROSS77a, ROSS77b]. They are further supported by the automated tools in PSL/PSA [TEIC77]. These techniques share a common feature of specification methods in the conventional model: they all focus on the external behavior specification of a target system and leave the internal structure of the system as a black box.

According to the specification, an overall system structure is first proposed as a preliminary design. It is then refined into more detailed modules, a top-down approach [WEGN79], subjected to the underlying implementation environment. That is, the design process tries to manage environmental resources to meet specified requirements. Note the output of the detail design process is the behavior specification of each module. The internal structure of each module is still a black box and left to the implementation stage.

Implementation phase turns the design specification into an implemented version using implementation languages. Either top-down or bottom-up approach can be applied to implement a system. Briefly, the bottom-up approach codes each module first followed by the integration of them as a whole. The top-down approach instead realizes the intermodule mechanisms by "studding" each module before the actual

implementation of each module is done.

Implemented codes have to be verified (to know that it works) and to be validated (to know that it works as users required) before being termed as a prototype. Testing is one of the commonly used techniques to debug the implemented codes. Since the testing process is performed on the implemented version, the most detailed level of the system, it becomes one of the most time consuming phases in the conventional model.

Maintenance and evolution is a stage to keep the implemented system working well and growing well as environment changes. The reason for that is the software development cost has become the dominant factor of the whole system development. A software system of evolutionary capability can mean a significant reduction of software development cost. In fact, as software becomes larger and larger the requirement of reusability, adaptability, and evolutionary capability becomes indispensable. However, these features are heavily dependent upon the methodologies and techniques used in a software development model (or used in each stage of a model if the model is life-cycled). Various techniques have been introduced into the conventional model to satisfy this trend. For instance, wide spectrum languages with the nature of easy transformation from specification phase to

design phase have been advocated to improve the adaptability of a solution system [RAMA86]. Kernel-based implementation facilitates the reusability. It also increases the degree of evolution by constructing a family of systems with the same kernel and (slight) different portions of modification.

2.2 Operational Models

The most salient feature of an operational model is that it incorporates a formal specification method which enables users to make a specification in internal operational structures [ZAVE84], so that the specification could be executed in the early stage of the development process. The external behavior required by users becomes implicit since it can only be exposed through the execution of the specification. Due to this executable nature of the specification, it can be used as a rapid prototype to speed up the software development process.

One common nature in this approach is that an automation-based support environment is required. With the help of this environment, users may execute the specification, verify the specification, and change the specification in the early stage of the development. That is, potential incompleteness, inconsistency, and ambiguities inside the rapid prototype can be solved earlier.

A general structure of the operational model could be

sketched as Figure 2.

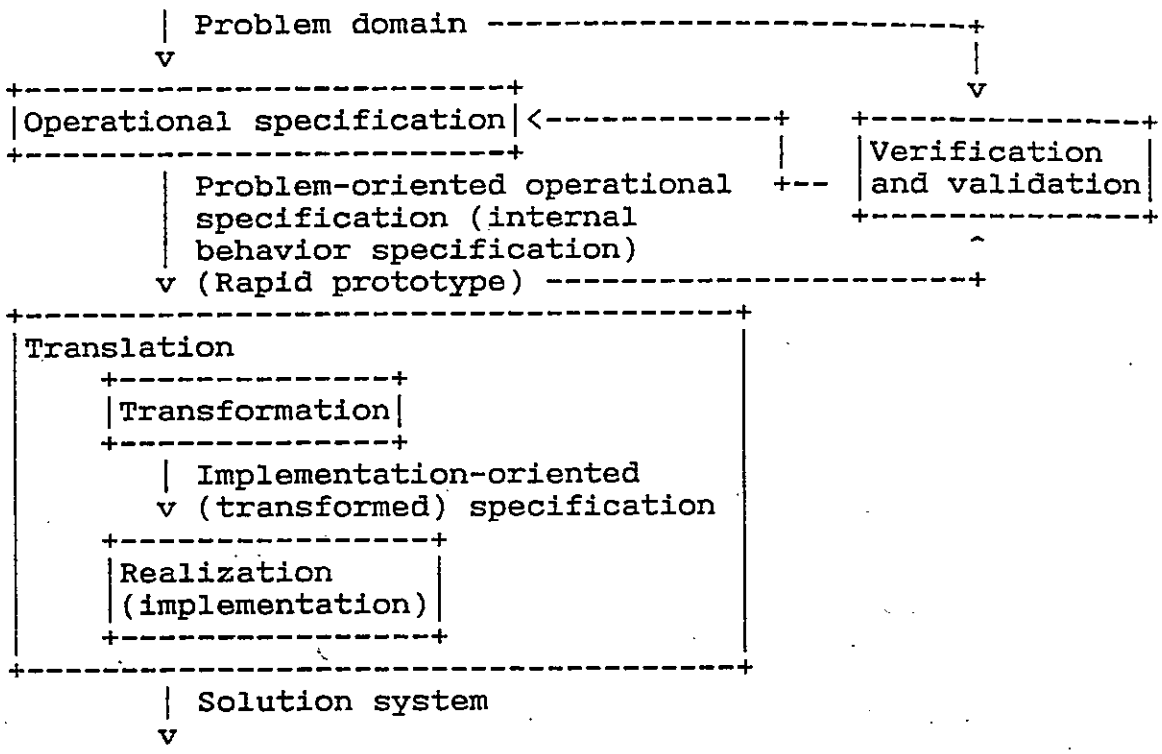

Fig. 2 Operational Model

Basically, the model consists of two major steps, namely, operational specification and translation. Operational specification deals with the description of the problem domain in a formal, internal-structure-based language. No implementation related factors, e.g., number of processors, their configuration, etc., are being considered during the step. This problem-oriented version of the specification (or a processed variant of the specification) will be used as the rapid prototype. Users then can run verification on the prototype against their own intent.

Translation refers to translating the specification into a real implementation. The best way to do this is via a compiler to translate the specification language into an implementation language automatically. However, it is quite difficult, since space of possible implementation is too large to search and the influence from local optimization on global optimization is still unknown [BALZ83]. These knowledge-intensive events normally need people to make high level decisions. This observation implicitly partitions the step into two substeps, namely, transformation and realization. Wherein, transformation takes into account the global strategies in implementing the system, e.g., selecting algorithm, control structure, data representation, and interfacing techniques under an existing environment, using buffering technique to improve performance, and so forth. This process will output an implementation-oriented (transformed) specification from the original problem-oriented specification. Computers then can translate the transformed specification into an implementation at the realization step and at the same time does some local optimization.

Many recent researches are concentrating on the development of operational specification languages using concepts from, for instance, functional programming [HEND86], Petri nets, or transition diagram. All of them are supported by appropriate automation tools for the verification process.

implemented as an "encapsulated" entity with associated operations.

Several features of the model deserve notice. First, objects associated with their operations are encapsulated. It enforces the information hiding and a limited visibility of objects. That is, the behavior of an object can only be decided by its response to the stimuli from environment. One consequence of that is the improvement of the degree of system modularity. Second, an object contains both activity and state of an entity. Thus it represents a complete entity of the real world. The correspondence between the model and the real world becomes more direct and more natural, which is one of the most attractive traits of the model [BORG85]. In general, the object-oriented model may help develop a software which is more comprehensive, more maintainable, more enhanceable, and more suitable for evolutionary enhancement.

Examples of object-oriented model starts with the very first commercialized system Smalltalk-80, which provides a programming environment as well as a language to facilitate the development of an object-oriented software [KRAS83]. [BOOC83] shows how to use Ada as a tool for object-oriented software developments. By surveying and comparing various object-oriented languages, [BYTE86] serves as a good intro-

duction to the concept of object-oriented model.

2.4 Knowledge-Based models

[BALZ83] argues that computer-based automation support for software development can overcome two shortcomings of conventional model, namely, poor manageability and maintainability. The former stems from the informal, large documentation during the development phases centered on the human aspect. The later comes from the maintenance on the source code level. One way to alleviate the situation is to shift from conventional human-centered process to computer-based approach. Thus, the central concept in computer-based automation support is to develop a knowledge-based assistant which works as a coordinating manager during the various phases of software development to facilitate the management and maintenance of the software [WEGN84]. The assistant itself is not constrained to any predefined model. Instead, it provides a framework for the automation of various types of software development model.

Applying this concept to the software development needs the development of the support environment first. It could be looked as a two-step process. Firstly, formal activities for each phase of a chosen model are set up. Then, a knowledge-based assistant is developed. The assistant will aid developers in performing and coordinating those activities,

and in recording the documentation of the system's development. Since many phases of the software development are knowledge-intensive activities, e.g., requirements specification, implementation refinement, etc., a suitable human-machine interface allowing the developers and the assistant effectively working together is necessary.

A general structure of knowledge-based model may contain following expert systems [FREN85]:

(a) A "Requirements Specification Assistant" to help the process of requirement acquisition and modification;

(b) Various "Activities" to perform specification simplification, consistency check, validation, explanation, and so forth;

(c) An "Implementation-oriented Decision Support" to help users define global implementation policies as well as local implementation optimization;

(d) A "Monitor" to measure performance, efficiency, and various statistics, e.g., execution frequency of program and data structures, which could be used to help optimization of the implementation;

(e) A "Development History Manager" to record changes to modules along with rationale for the changes;

(f) A friendly "User Interface" to support the activity of sophisticated explanation and tutoring;

(g) A "Coordinator" serving as a glue to invoke and coordinate all modules of the support environment.

3. A Development Methodology for Knowledge-Based Office Systems

Last section shows the progressive history of software development models. It implicitly brings out the pros and cons of each software development model. For instance, we realize that operational models seem suitable for rapid prototyping. However, to directly specify the whole internal structure of a large system is quite a labor-intensive job [ZAVE84]. Since the concept of "processes" used in operational models is only a "projection" of system "objects" [YEH84], we can employ the object-oriented decomposition technique to alleviate the situation.

[BORG85] advocates that the use of symbols and definitions in a model should correspond to concepts and entities in the real world. The structure of the model should also mirror the structure one perceives in the real world. Such a "world-oriented" concept in fact speaks for the society model we developed for office information systems. Recall the components used in the society model. They are carefully selected and termed to reflect entities and structure of real world offices.

As we pointed out in [HO87], the society model combines

the concepts from object-oriented approach and knowledge-based approach. (Imagine office agents as system objects and represent objects as knowledge-based entities.) Some features accruing from the combination of these approaches are:

(a) A direct, natural correspondence between the modeled system and the real world office can be established. Any further decomposition of office agents may follow object-oriented approach;

(b) System objects are viewed as knowledge-based entities. Thus, "knowledge engineering" techniques can be applied to the development of these knowledge-based systems.

In general, knowledge engineering, which is concerned with the the development of a knowledge-based system, refers to following activities [FEIG78]:

(1) Understanding the problem task (same as conventional programs development);

(2) Selecting suitable knowledge-based systems development tools (e.g., program organization, inference methods, knowledge representations, etc.) for the task;

(3) Eliciting knowledge from users and organizing it for use by the program;

(4) Refining and reconceptualizing the system if system performance degrades too much as knowledge in-

creases;

(5) Making interface comfortable to users and system behavior understandable and controllable by users.

The difference between the technique and general software engineering techniques stems from the difference between a knowledge-based system and a general software program. Note that all knowledge except some "control" information is extracted from conventional programs and treated as another set of "data" (in fact, knowledge) in a knowledge-based system [KOWA79]. That makes tools selection and knowledge attribution two "critical" processes [FEIG78]. By which, we mean these two processes will profoundly affect the performance of a knowledge-based system.

It is our belief that if equipped with suitable tools, the agony of transfer of expertise (the process of eliciting knowledge from users and attributing it into the program) can be relieved. Knowledge table systems developed in [CHAN86a], which include a unified knowledge representation method, different knowledge inference models, knowledge editing tools, etc., can serve this purpose. By the employment of the unified representation method provided by the knowledge table system, knowledge of a whole office system can be prototyped through a single tool, which further speeds up the system development.

This retrospection leads us to forming a "mixed" methodology for the development of a knowledge-based office system:

(1) Use the society model as a world-oriented model for office information systems to define and specify an office as a set of office agents;

(2) Use object-centered decomposition technique to refine office agents to more primitive objects if necessary (e.g., an agent refined to a set of ks's);

(3) Primitive objects are then prototyped as knowledge table systems [DOYL85]. As a matter of fact, accommodating other advantageous models, e.g., operational models, at this step for the rapid prototyping of primitive objects is as natural as we have done with knowledge table systems;

(4) Set up a knowledge-based support environment to support various aspects of the development process, i.e., system specification, verification, evaluation, and optimi-zation (see below for detail).

Figure 3 depicts such a methodology, which is derived from the methodology we described in [CHAN86b] for the development of distributed knowledge-based information systems.

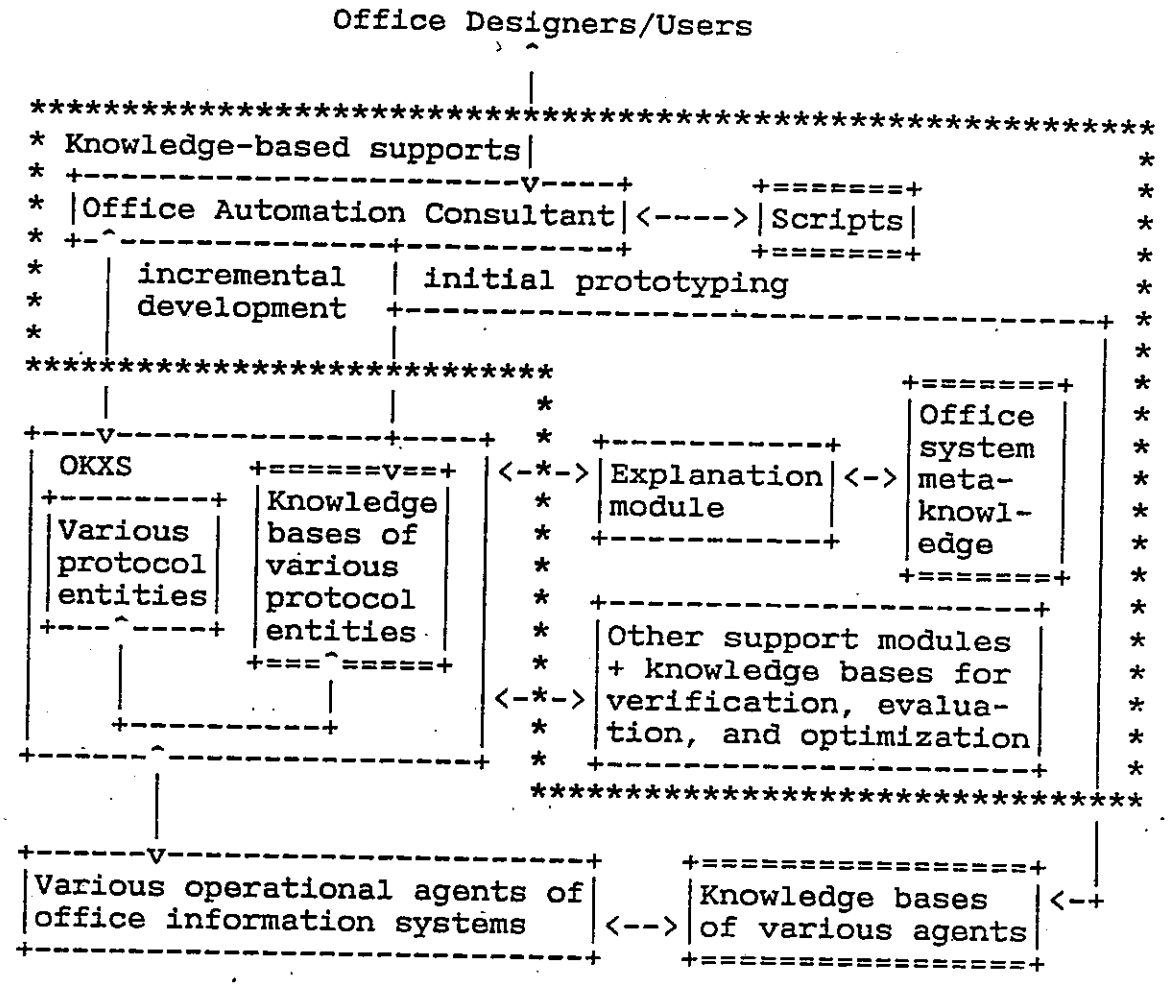

Fig. 3 A Methodology for the Development of a Knowledge-Based Office System

Note that the knowledge-based model is adopted to provide a knowledge-based support environment for the development process. It is shown in the figure as an asterisk enclosure. The target office system is developed as various office agents interacting through an Office Knowledge Exchange System (OKXS) according to the society model. To emphasize the knowledge-based nature of the office system,

all modules shown in the figure are accompanied with associated knowledge bases. That includes the OKXS system which is characterized as a set of protocol entities and their associated knowledge bases, which virtually provide the office knowledge exchange protocols. This approach thus unifies the design of the whole system.

Two components of the knowledge-based support system are singled out in Figure 3, namely, Office Automation Consultant (OAC) and explanation module. OAC is a knowledge-based environment working as the interface between office users and the office system. It is the one which incrementally sets up the whole system. In the initialization stage it is used to transfer knowledge from office users to various components of the office system. It also works as an aid for the verification and the enhancement phases of the system. A script of how to acquire knowledge from users, how to adequately attribute knowledge into each agent without causing conflict, how to invoke other modules for the optimization of the system, etc. is associated with the consultant as its knowledge base.

Explanation module is responsible for constructing and explaining lines-of-reasoning of individual agent or a whole office system to relevant users. It has knowledge of how the office is configured, how knowledge is structured and processed in the knowledge table system, etc., (termed as

office system meta-knowledge) to facilitate the explanation process. Users may understand the system operation and debug system knowledge by following the lines-of-reasoning. Other modules related to the verification, evaluation, and optimization of the office system can be introduced into the environment for further improvement of the support system.

In general, OAC takes part in all aspects of the office system development process:

(a) Specification: In the initialization stage, OAC provides an office knowledge specification script (based upon the society model) for office users to prototype each office agent including the OKXS system. During the enhancement (or evolution) phase, new specification can be incorpo-rated into the system by OAC through OKXS;

(b) Verification: Running a system with new posted knowledge can be done by knowledge table interpreter invoked by OAC, which provides a quick feedback for verification. Explanation module plays an important role in this phase. Through its exposition of system reasoning scenario, the verification turns out to be easier. In addition, a variety of verification tools can be called by OAC during this phase. For instance, correctness check on input specification supported by specification checker,

OKXS protocols verification, knowledge tables verification, agents' knowledge verification, etc;

(c) Evaluation: OAC can invoke suitable modules to perform system analysis capabilities such as static evaluation, dynamic simulation, and so on;

(d) Optimization: System global optimization such as system configuration, system knowledge distribution, agent knowledge partition, and so forth, can be achieved by proper modules invoked by OAC.

Thus, OAC works as a coordinator of the support system. It coordinates specific modules in the support environment to support specific capabilities for the development of a knowledge-based office system. In short, it works as a tool for developing a knowledge-based office system by rapidly prototyping office system knowledge or as a tool for the evolution of an office system.

4. Prototyping OAC as a Knowledge Table System

One interest point of the methodology is that the supporting environment can be developed using the same technique. For instance, OAC can be prototyped as a knowledge table system using knowledge table editor "kted" described in [HO87]. Once OAC is developed, the whole system development can be bootstrapped since most office agents are prototyped through OAC.

Developing OAC as a knowledge table system may serve as a pragmatical example of how we prototype a knowledge-based entity using kted. The very first step is the identification of all knowledge that makes OAC work. Kted is then used to define these knowledge as various knowledge tables. The following is a list of knowledge that OAC is expected to consist of:

(a) Knowledge of the society model and how the mixed strategy is applied to the office systems knowledge acquisition process;

(b) Knowledge of consistency and/or completeness check of acquired office knowledge;

(c) Knowledge of other modules which are responsible for various verification, evaluation, or optimization functions as well as knowledge of when to invoke relevant modules.

A script is designed for OAC which incorporates all these knowledge. Basically, the script uses the society model as a guidebook to seduce knowledge for each component of the model and uses object-oriented approach to refine the specification wherever office agents are encountered. The script also takes into account the verification, evaluation, and optimization during or after the specification phase. Thus, consistency and completeness check on the specifica-

tion can be performed on the fly or when required from users. An English description of the script is given in Appendix A.

The script consists of four acts, and each act in turn contains several scenes. Although each scene gives a sequential listing of actions that the OAC has to follow, scenes are not necessarily connected sequentially. A specific act or scene is entered when its situation is satisfied. Acts or scenes might be invoked by demon processes during the execution of other acts or scenes.

Now, we are ready to prototype the script into a knowledge table system. Several ways to implement such a script. For instance, we may have an Ic-type table to group four acts together. Each entry of the table then links to another Ic-type of knowledge table describing each scene. Each scene can be represented as an Fc-type table. Another way is to directly implement each act as a separate Ic-type of knowledge table. Thus, we will have four separate invocation alerters to invoke them respectively. Since we expect communication between acts may exist, we prefer the first method, which provides a whole view of OAC knowledge. Through the kted, we can transform the script to knowledge tables and store them into the database. Appendix B shows part of the script knowledge tables.

The first run of the script will produce a prototype of the target office. In fact, the prototype contains various knowledge tables to represent knowledge of the target office. Next runs of OAC will modify the target office through kted. Appendix C shows object definitions of some of these knowledge tables. Some modules are invoked by OAC for various purposes, e.g., verification, evaluation, and optimization. Object definitions of knowledge bases of some of these modules are included in Appendix C to help understand knowledge tables of OAC script.

5. Summary

A mixed methodology has been developed for the development of a knowledge-based office system. The methodology bases itself on the society model, which inherits merits from techniques of object-oriented approach and knowledge-based approach. Thus, the methodology may help develop a software system which naturally mirrors the real world counterpart. The specification of a whole system also becomes more tractable due to the object-oriented decomposition technique.

The knowledge-based approach impacts the methodology in two aspects. First, it introduces the knowledge engineering techniques into the general framework of software engineering. The techniques emphasize the knowledge acquisition

process as well as the system behavior explanation aspect, which facilitates the knowledge prototyping as well as the knowledge management of a knowledge-based system. By adopting knowledge table systems as a unified representation method for the storage of the target system knowledge, the methodology may further speed up the system development process.

The knowledge-based approach also influences the design of the development tools. A knowledge-based supporting environment is incorporated into the methodology to help manage the whole system development process. Among them, OAC is designed as a coordinator of the supporting system. It works as a knowledge prototyping tool for the target office during the system initialization and evolutional phases. A system behavior explanation module is included in the environment serving as an interface for the understanding of the system behavior as well as a debugging tool for the system knowledge.

Once OAC is developed, the whole office system can be prototyped through OAC. That includes the OKXS system, which is uniformly treated as a set of knowledge-based protocol entities. The prototyping process of a target office system follows the script of the OAC, which is more or less like what we used to prototype the OAC as a knowledge table system.

Accommodating operational models inside the framework of the mixed methodology is quite natural due to the generality of the society model. [CHAN86a] is an example, which adopts the OPM model, rather than the micro society structure, to model office agents as a procedural description of their activities. Since the knowledge table is a unified representation method, the knowledge description in the OPM model can be represented as a knowledge table system, too.

Appendices

Appendix A:

The English version of the OAC script

ACT1: System specification:

<situation (specification (office-environment goal missions regulations objectives facilities agents structures activities))>

<demon: A verification on the environment will be run through ACT2/Scene2>

Scene1: Office specification.

<demon: Agents specified in this scene have to be refined in Scene2>

1. Acquire office goals (office->goals);
2. Acquire structures which might refine some of office goals (goal->subgoals);
3. Acquire structures related to all (sub) goals (goal->agents);

   /** Note that office facilities are treated uniformly as office agents. **/
4. Acquire other structures if any;
5. Acquire applicable regulations;
6. Setup new protocol entities for OKXS (OKXS->agents).

```
                /** Protocol entities are treated as
                    agents too.
                 **/
        Scene2: Agent specification.
                <demon: Agents to be refined have been de-
                 fined in Scene1?>
                1. Acquire agent goals (agent->goals);
                2. Acquire ks "agent-descriptor";
                3. Acquire structures related to all agent
                   goals (goal->ks's);
                4. Acquire other structures if any;
                5. Acquire ks "micro-planner";
                6. Acquire each activity ks.
ACT2: System verification.
      <situation (verification (explanation consistency-
       check completeness-check testing debugging))>
        Scene1: Explanation.
                1. Office-based explanation.
                   1.1 Answer office static knowledge query;
                       /** For instance, statistics of some
                        office items.
                        **/
                   <demon: Need to run ACT3/Scene1?)>
                   1.2 Explain office reasoning.
                       /** For instance, Why this procedure
                           is used (or formed) by office
```

agents to deal with that office goal?

**/

<demon: Rerun a case (e.g., executing an office procedure)>

2. Agent-based explanation.

2.1 Answer agent-based static knowledge query;

/** For instance, agent expertise.

**/

<demon: Need to run ACT3/Scene1?)>

2.2 Explain agent reasoning.

/** For instance, Why the agent takes this action instead of another to complete that subgoal?

**/

<demon: Rerun a case>

Scene2: Consistency check.

1. Consistency check on "office->agent->ks's" decomposition;

2. Consistency check on each office structure;

/** For instance, poor labor match in community structure, reflective relation "report-to", etc.

**/

3. Consistency check on each micro structure of each agent;

/** For instance, agent role conflict. **/

4. Consistency check on knowledge conflict and subsumption on office level as well as agent level;

5. Consistency check on OKXS communications using protocol verification techniques.

Scene3: Completeness check.

1. Completeness check on agents delegation;

2. Completeness check on ks's delegation;

3. Completeness check required and performed by other modules.

Scene4: Testing.

<default: Run a random case (i.e., system chooses a procedure)?>

1. Acquire test target or default;

2. Run Scene1 and/or 2 and/or 3;

3. Debug.

ACT3: System evaluation.

<situation (evaluation (static-evaluation dynamic-simulation))>

Scene1: Static evaluation.

1. Acquire evaluation model, evaluation target, and relevant parameters;

<default: per procedure with predefined parameters>

/** Examples of evaluation target: agent, procedure, community, or office based query; data retrieval frequency; turn-around time. Examples of parameters: time, probability.

**/

2. Invoke relevant evaluation module.

Scene2: Dynamic simulation.

1. Acquire simulation model, simulation target, and simulation parameters;

/** Examples of simulation target: agent, procedure, community, or office. Examples of parameters: load distribution, simulation-time-scale.

**/

2. Invoke relevant simulation module.

ACT4: System optimization.

<situation (optimization (procedure-streamlining structure-reconfiguration local-optimization global-optimization))>

Scene1: Procedure streamlining

<demon: Structure optimization will be run

by Scene2>

1. Acquire a target procedure to be streamlined;
2. Acquire new or temporary constrains if any;
3. Invoke relevant streamlining module to advise a new structure for the procedure which will make better use of the system resources.

Scene2: Structure reconfiguration

1. Acquire the target to be restructured;
   /** A community or an office.
   **/
2. Acquire new or temporary constrains if any;
3. Invoke relevant configuration module to advise a new configuration for the target so that the performance of the restructured target meets users' requirement.

Appendix B:

Knowledge Tables Representing OAC Script

Table "script"

table-name: script	
control-condition: office consultation	I-type

	factor-obj1
obj-name	NIL

	inference-id	exp	cons
obj-name	act-name	situation	action
obj-type	VAL[20] %	EXP[200]	KT[50]

% Notation of "obj-type[m]" represents that the type of the object is "obj-type" described at most in m characters long.

act-name->20	situation->200	action->50
specification	(specification*	link(specification)
verification	(verification*	link(verification)
evaluation	(evaluation*	link(evaluation)
optimization	(optimization*	link(optimization)

(specification*=(specification (office environment goal missions regulations objectives facilities agents structures activities))

(verification*=(verification (explanation consistency-check

completeness-check testing debugging))

(evaluation*=(evaluation (time probability static dynamic simulation))

(optimization*=(optimization (streamlining reconfiguration optimization))

Table "specification"

table-name: specification	
control-condition: linked	I-type

	factor-obj1
obj-name	NIL

	inference-id	exp	cons
obj-name	scene-name	situation	action
obj-type	VAL[20]	EXP[200]	KT[50]

scene-name->20	situation->200	action->50
office-definition	(office*	link(office)
agent-definition	(agent*	link(agent)

(office*=(office environment goals missions regulations objectives facilities agents structures knowledge-exchange-protocols)

(agent*=(agents micro-structures activities knowledge-sources)

Table "office"

table-name: office	
control-condition: linked	F-type

	obj1	obj2
obj-name	step-name	step-action
obj-type	VAL[30]	ACT[100]

step-name->30	step-action->100
goals-spec	eval(kted office-goals)
g-refinement-spec	eval(kted goal-subgoals)
g-structure-spec	eval(kted sub.goal-agents)
o-structure-spec	eval(kted new-table)
regulation-spec	eval(kted regulations)
OKXS-new-entities	eval(kted OKXS-agents)

Table "agent"

table-name: agent	
control-condition: linked	F-type

	obj1	obj2
obj-name	step-name	step-action
obj-type	VAL[30]	ACT[100]

step-name->30	step-action->100
goals-spec	eval(kted agent-goals)
agent-descriptor	eval(kted agent-descriptor)
g-structure-spec	eval(kted goal-ks's)
o-structure-spec	eval(kted new-table)
micro-planner	eval(kted micro-planner)
each-activity-ks	eval(kted new-table)

Appendix C:

Knowledge Tables Produced/Used by OAC Script (table type followed by a list of object names shown within parentheses)

(i) Office definition

i.1 office-goals

(F-type: office goals)

i.2 goal-subgoals

(F-type: goal subgoals)

i.3 goal-agents

(F-type: sub.goal agents)

i.4 regulations

(I-type: rule-sets)

i.5 OKXS-agents

(F-type: OKXS agents)

(ii) Agent-definition

ii.1 agent-goals

(F-type: agent goals)

ii.2 agent-descriptor

(F-type: agent-static-attr role acquaintance-rel)

ii.2.1 agent-static-attr

(F-type: agent-id agent-name age sex birthday expertise)

ii.2.2 acquaintance-rel

(F-type: responsible-for report-to commands familiar-list)

ii.3 goal-ks's

(F-type: goal ks's)

ii.4 micro-planner

(I-type: conditions actions)

(iii) Knowledge-table-alerters

iii.1 alerters

(I-type: rule-id conditions creator date actions comment)

(iv) Rules for various check/verification

iv.1 office-check-rules

(I-type)

References

[ALFO77] Alford, M., "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Trans. Software Engineering SE-3 (Jan. 1977), 60-68.

[BALZ83] Balzer, R., T. E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer 16 (Nov. 1983), 39-45.

[BELL77] Bell, T., D. Bixler and M. Dyer, "An Extensible Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. Software Engineering SE-3 (Jan. 1977), 49-60.

[BRUN86] Bruno, G. and G. Marchetto, "Process-Translationable Petri Nets for the Rapid Prototyping of Process Control Systems," IEEE Trans. Software Engineering SE-12, 2 (1986), 346-357.

[BOOC83] Booch, G., Software Engineering with Ada (Benjamin/Cummings, CA, 1983).

[BOOC86] Booch, G., "Object-Oriented Development," IEEE Trans. Software Engineering SE-12, 2 (1986), 211-221.

[BORG85] Borgida, A., S. Greenspan, and J. Mylopoulos, "Knowledge Representation as the Basis for Requirement Specifications," IEEE Computer 18 (April 1985), 82-91.

[BRUN86] Bruno, G. and G. Marchetto, "Process-Translationable Petri Nets for the Rapid Prototyping of Process Control Systems," IEEE Trans. Software Engineering SE-12, 2 (1986), 346-357.

[BYTE86] BYTE, "Object-Oriented Languages," Theme topic of BYTE 11, 8 (Aug. 1986).

[CHAN86a] Chang, S. K. and C. S. Ho, "Knowledge Table As a Unified Knowledge Representation Method," Illinois Institute of Technology Technical Report (IL, 1986).

[CHAN86b] Chang, S. K., C. S. Ho, C. Y. Hsieh and L. Leung,

"A Methodology for Distributed Knowledge-Based Information System Design," In Proc. IEEE Workshop Language for Automation (Singapore, Aug. 1986).

[DAVIS77] Davis, C. G. and C. R. Vick, "The Software Development System," IEEE Trans. Software Engineering SE-3 (Jan. 1977), 69-84.

[DOYL85] Doyle, J., "Expert Systems and the "Myth" of Symbolic Reasoning," IEEE Trans. Software Engineering SE-11, 11 (Nov. 1985), 1386-1390.

[FEIG78] Feigenbaum, E. A., "The Art of Artificial Intelligence - Themes and Case Studies of Knowledge Engineering," In Proc. AFIPS NCC Vol. 47 (June 5-8, Anaheim, CA, 1978), 227-240.

[FREN85] Frenkel, K. A., "Toward Automating the Software-Development Cycle," Commun. ACM 28, 6 (June 1985), 578-589.

[HEND86] Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping," IEEE Trans. Software Engineering SE-12, 2 (1986), 241-250.

[HO86] Ho, C. S., Y. C. Hong and T. S. Kuo, "A Society Model for Office Information Systems," ACM Trans. Office Information Systems 4, 2 (April 1986), 104-131.

[HO87] Ho, C. S., Society Model Based Office Information Systems (PH. D. Dissertation, National Taiwan University, May 1987).

[JONE76] Jones, M. N., "HIPO for Developing Specifications," Datamation (March 1976), 112-125.

[KOWA79] Kowalski, R., "Algorithm = Logic + Control," Commun. ACM 22, 7 (1979), 424-436.

[KRAS83] Krasner, G., ed., Smalltalk-80: Bits of History, Words of Advice (Addison-Wesley, Reading, MA, 1983).

[KUO85] Kuo, T. S., et al., "An Agent Society Model for Office Information Systems," Institute of Information Science, Academia Sinica Technical Report TR-85-005 (Taiwan, Sept. 1985).

[RAMA86] Ramamoorthy, C. V., "Issues in Software Engineering

for Automation," Keynote Speech in IEEE Workshop on Language for Automation (Singapore, Aug. 1986).

[RAMA87] Ramamoorthy, C. V., S. Shekhar and V. Garg, "Software Development Support for AI Programs," IEEE Computer 20 (Jan. 1987), 30-40.

[ROSS77a] Ross, D., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. Software Engineering SE-3 (Jan. 1977), 16-33.

[ROSS77b] Ross, D. and K. E. Schoman, Jr., "Structured Analysis for Requirement Definition," IEEE Trans. Software Engineering SE-3 (Jan. 1977), 6-15.

[TEIC77] Teichroew, D and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Systems," IEEE Trans. Software Engineering SE-3 (Jan. 1977), 41-48.

[WASS86] Wasserman, A. I., P. A. Pircher, D. T. Shewmake and M. L. Kersten, "Developing Interactive Information Systems with the User Software Engineering Methodology," IEEE Trans. Software Engineering SE-12, 2 (1986), 326-345.

[WEGN79] Wegner, P., Research Directions in Software Engineering (The MIT Press, MA, 1979).

[WEGN84] Wegner, P., "Capital Intensive Software Technology. Part 2: Programming in the Large," IEEE Software (July 1984), 24-32.

[YEH84] Yeh, R. T., P. Zave, A. P. Conn and G. E. Cole, Jr., "Software Requirements: New Directions and Perspectives," In Handbook of Software Engineering (C. R. Vick and C. V. Ramamoorthy, eds., Van Nostrand Reinhold, NY, 1984).

[ZAVE84] Zave, P., "The Operational vs the Conventional Approach to Software Development," Commun. ACM 27, 2 (1984), 104-118.

[ZAVE86] Zave, P. and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," IEEE Trans. Software Engineering SE-12, 2 (1986), 312-325.