THE KASE APPROACH TO DOMAIN-SPECIFIC SOFTWARE SYSTEMS

Sanjay Bhansali and H. Penny Nii

Knowledge Systems Laboratory
Stanford University
701 Welch Road, Bldg. C, Palo Alto, CA 94304
bhansali@sumex-aim.stanford.edu
nii@sumex-aim.stanford.edu

1. Introduction

Designing software systems, like all design activities, is a knowledge-intensive task. Several studies, (e.g. [Adelson & Soloway, 1985; Guindon, Krasner, & Curtis, 1987]) have found that the predominant cause of failures among system designers is lack of knowledge – knowledge about the application domain, knowledge about design schemas, knowledge about design processes. The goal of domain-specific software design systems is to explicitly represent knowledge relevant to a class of applications and use it to partially or completely automate various aspects of the design activity for designing systems within that domain. The hope is that this would reduce the intellectual burden on the human designers and lead to more efficient software development.

In this paper, we present a domain-specific system built on top of KASE, a knowledge-assisted software engineering environment being developed at the Stanford Knowledge Systems Laboratory. We introduce the main ideas underlying the construction of domain specific systems within KASE, illustrate the application of the idea in the synthesis of a system for tracking aircrafts from radar signals, and discuss some of the issues in constructing domain-specific systems.

2. Domain Specific Software Systems using KASE

KASE is a knowledge-based software development environment that is designed to provide *active* assistance in the design of software systems. Some of the basic characteristics of the KASE environment are: a domain-independent representation mechanism for software architectures, a graphical interface that permits smooth navigation between different views of a software system [Guindon, 1992], an integrated editor that permits modifications to the architecture from any view, and a constraint checker that can help a user maintain various syntactic and stylistic constraints between different components of the architecture[Nii, Aiello, Bhansali, Guindon, & Peyton, 1991].

The construction of domain specific software systems in KASE involves the identification of a *generic problem* or task, a *generic architecture* suitable for the task, a model of the application domain in terms of primitive entities (e.g. object, relations, events), and a set of *customization tools* that can be used to construct a specific system for a particular problem.

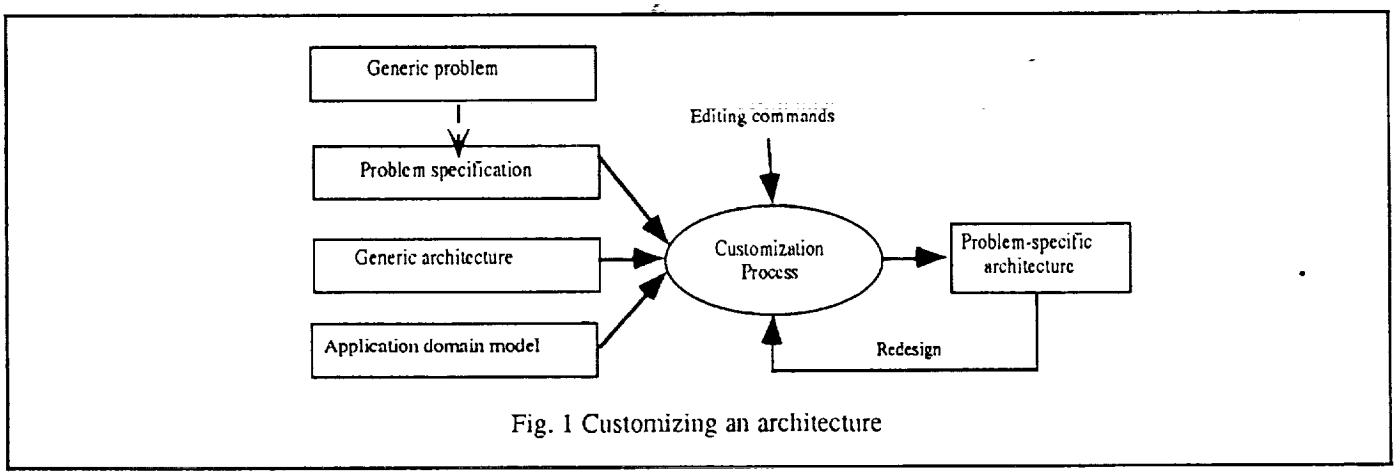

Fig. 1 Customizing an architecture

As shown in figure 1, the software design activity consists of instantiating the generic architecture with respect to a given problem statement and the domain model using the customization tools and results in the creation of a problem-specific architecture. We call this process *customization* - customize a generic architecture to fit an application.

A generic problem represents a class of problems. By identifying problem classes, one can design knowledge representation schemes, architectures, and reasoning processes which are appropriate for the general problem, and reuse them for several different problem instances. The specification of a generic problem results in the creation of a problem schema which specifies the high-level structure of a problem specification. A schema has certain *roles* which represent the parameters of the problem, and *constraints* on the values of the roles. Instantiating these roles with specific values results in the creation of a specific problem specification.

Figure 2 shows the schema for an example generic problem: tracking a set of mobile objects by interpreting signals that are being continually generated by the objects. (This generic problem can be instantiated, e.g. to the problem of tracking aircrafts from radar and voice signals (Brown, Schoen, & Delagi, 1986) or tracking ships from sonar data (Nii, Feigenbaum, Anton, & Rockmore, 1982)). This problem has three parameters: (i) the specification of the input signal(s); (ii) the main body or functional description of the problem in the form of an extremely high-level program; and (iii) certain characteristics of the domain and the environment. The constraints on the schema roles are specified by specifying a grammar for instantiating the roles.

Associated with each generic problem is a set of (possibly one) generic architectures, which can be used to create a system for solving instances of the generic problem. A generic architecture is a collection of *parameterized modules* and intermodular dependencies. A parameterized module is a logical collection of software entities like procedures, types, etc. in which some of the entities are abstracted as parameters. A parameter can be, among other things, an algorithm, a representation scheme, or a sub-module. The design process is viewed as an instantiation of the various parameters comprising a generic architecture. However, the parameters can be fairly complex entities and the design task is non-trivial.

The structure of the generic architecture determines the basic solution strategy for solving the problem. For example, the continuous signal interpretation problem given earlier can be solved using a symbolic, knowledge based approach, or by statistical analysis of the data and the two solutions would have radically different architectures. A module description includes information about the input and output data flows of the module, the submodules/supermodules structural relations, the services it requires from other modules, the services it provides to an external module, the precondition and postconditions for each service provided by the module, and/or a program template that implements each service. The most interesting aspect of the module description is that some of its attributes are viewed as parameters of the module. Associated with each parameter attribute is a method which can be used to determine the value of the parameter. The complexity of the method depends on the type of the parameter. For example, it may be a simple process of selecting between a pre-determined list of alternatives, or it may involve sophisticated reasoning using domain knowledge and heuristic rules.

Continuous-Signal-Interpretation :Generic-problem
Signal-Inputs:)<var> : (SEQ :FROM <int> :TO <int> (<fields>)
<field-description>)]+
Body: WHILE <formula> DO <statements> ENDWHILE
Task Assumptions: <task-assumptions>
where
<fields> ::= <identifier> | <identifier> <fields>
<field-description> ::= EXIST <objects> SUCH-THAT <condition>
<statements> ::= <statement> ; <statements> | <statement>
<statement> ::= (IF <formula> THEN-DO <statement>) |
(FORALL <vars> <formula> DO <statement>) |
(PRINT <terms>)
<task-assumptions> ::= (UNRELIABLE-SIGNAL <var>) |
(REDUNDANT-SIGNAL <var>) |
(ASYNCHRONOUS-SIGNAL <var>) |

Fig. 2. Specification of the generic problem of continuous signal interpretation.

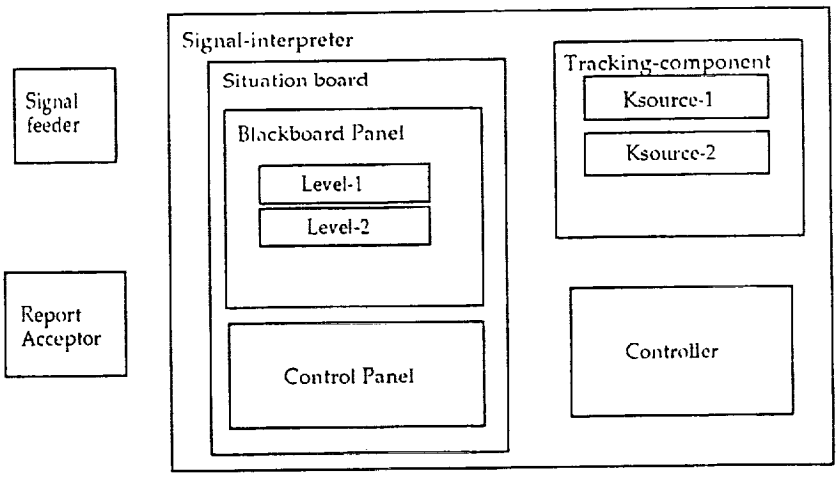

Fig. 3. A functional decomposition of the generic architecture for the continuous-signal interpretation problem. The architecture shows the main modules comprising the architecture.

Signal-Interpreter **isa** module

submodules	Situation-board, Tracking-component, Controller
supermodule	CSI-system
inputs	?s : SEQ(signal)
outputs	?r : SEQ(report)
requires	(print-report ?r), (read-next-signal) :signal, (start-execution)
provides	(main)
calls	report-acceptor, signal-feeder
called-by	nil
parameters	1) Controller 2) SituationBoard
constraints	1) Controller is instantiated to an EventDriven-Controller iff SituationBoard is instantiated to an EventDriven-SituationBoard. 2) Only the TrackingComponent should have a dataflow into the SituationBoard. 3) Only the Controller module can call the Tracking-Component. .....

Fig. 4. Representation of the Signal-interpreter module in the generic architecture.

Figure 3 shows the structural decomposition of a generic architecture for the continuous-signal-interpretation problem class and figure 4 shows a partial description of the signal-interpreter module of the generic architecture.

The domain model provides the ontology of terms and operations used to describe an application domain independent of a specific task; several different problems can be specified in a high-level language using this ontology. The primary components of the domain model are *objects* and *relations* between the objects. An object is an abstraction of some entity in the application domain, e.g., an aircraft or a signal. Associated with each object is a set of *attributes* which are properties that describe an instance of an object and *operations* that change the state of an object. The description of an operation includes pre- and post-conditions and optionally, a code template that implements the operation.

2.1 CUSTOMIZATION PROCESS

The customization process consists of refining a selected generic architecture into a detailed architectural specification based on the model of the domain and the problem specification. In KASE, the customization process is performed in an interactive and mixed-initiative setting. The role of KASE in the design process is that of an intelligent

design associate that provides suggestions on how to refine the architecture, carries out the commands invoked by the user, informs the designer of constraint violations in the design, keeps a record of the design steps and the dependencies between the steps so that incremental modifications to the design can be done efficiently.

The knowledge used by KASE in providing these kinds of assistance includes general, domain independent knowledge about software design, architecture-specific knowledge for the instantiation of various architectural parameters, as well as specific heuristic knowledge about design related to a particular domain. Most of the domain independent design knowledge is represented in the form of constraints (e.g. those relating different levels of a data flow diagram), and KASE contains mechanisms which automatically keep track of these constraints as well as heuristics for resolving constraint violations (Nii *et al.* 1991). The architecture specific knowledge includes a set of constraints governing the relationships between different components of the architecture, a library of reusable modules and schemas which can be used to instantiate the architectural parameters, and a collection of design rules and procedures that can be invoked by a designer to instantiate certain parameters and optimize the design.

To illustrate the customization process, consider the generic architecture shown in fig. 2. The parameters in the generic architecture include the following: 1) the submodules of the blackboard panel, 2) the type of information stored in the control panel, 3) the submodules of the tracking component, and 4) the scheduling and focusing strategies of the controller. Different instantiations of these parameters result in the creation of a widely different systems with different performances. KASE contains a set of design rules for instantiating these parameters, and a set of transformation rules that optimize the design (e.g. merging certain kinds of control signals into one for increased efficiency). The customization process for an implemented example in KASE is described in [Bhansali & Nii, 1992].

2.2 REDESIGN

Software design is characterized by frequent modifications either due to a design error or as a result of a change in the problem requirements or the computing environment. KASE uses different mechanisms to support these two kinds of modifications.

2.21 Redesign due to error in original design. KASE automatically checks for violations of several kinds of constraints and helps the designer modify the architecture to resolve the inconsistencies. The constraints in KASE are currently divided into three categories: 1) General architectural constraints (e.g. every data link must have a consumer and a producer); 2) Specific architectural constraints (e.g. there must be no data flow or control flow between submodules of the tracking component); and 3) Stylistic constraints that are derived from design principles that are considered 'good' (e.g a module must not be decomposed into more than *n* submodules at any level of abstraction).

Each constraint in KASE is associated with a *trigger*, a *predicate*, and an optional *resolving-action*. A trigger is a set of actions that can potentially cause the constraint to be violated, a *predicate* is a Lisp expression that checks to see whether the constraint is actually violated, and *resolving-action* is a set of actions that may be taken to remedy the constraint violation. KASE monitors the design activity and flags each constraint that is triggered by a user action. When a user indicates the completion of a design session, KASE checks the predicates for each flagged constraint to see whether the constraint is actually violated. Quite often, a constraint that gets violated by a design action is resolved by a later action, and such constraint violations should be, and are, transparent to the designer.

When KASE reports a constraint violation, the designer can ask KASE for a list of suggestions on how to resolve the error. Depending on the nature of the constraint, KASE presents a list of different actions that may be taken to remove the constraint violation. The user can then choose either one of the actions suggested by KASE or take some other action.

2.2.2 Redesign due to change in requirements. KASE provides tools that can help a designer in modifying parts of a design to meet new requirements without having to start from scratch. First, KASE maintains a history of all the design steps and allows the user to go back to any previous state of the design. It does this by replaying the design history from the initial state to the desired state.

A second redesign support provided by KASE is in localizing the effects of a design change. KASE uses dependencies between design steps to structure a linear design history into a lattice. When the user wants to undo the effect of a particular design step, KASE uses the position of that design step in the derivation history to determine what other design steps are affected by it [Bhansali, 1992].

3. Discussion

In this section we briefly discuss some of the issues, advantages, and limitations in our approach. One of the major issue in the design of domain-specific systems is concerned with acquiring and maintaining the extensive body of knowledge from multiple sources. This task, also known as domain modeling, is a manifestation of the classic

knowledge acquisition problem in expert systems. One way of viewing generic problems/tasks and architectures is to consider them as providing a skeletal knowledge base or shell which can be instantiated for different applications. Our long term goal is to provide a library of generic problems and associated architectures, which would provide a base from which various domain models can be instantiated.

A second issue is concerned with the flexibility of the resulting system. Domain specific systems utilize specialized design techniques which are well suited for a particular class of applications. However, since it is not possible to anticipate all subsequent changes in requirements, the specialised design techniques may not be adequate for extending the system beyond the original intended application. A major effort in the KASE project has, therefore, been expended in providing a domain--independent infrastructure which enables a user to modify an architecture through an integrated editor, pictorial and symbolic visualizations of the design from various perspectives, and a constraint maintenance subsystem that supports opportunistic design based on insights drawn from empirical studies of human designers [Guindon, 1990].

A third issue is concerned with the usefulness of the approach. Our approach involves a considerable investment in terms of building the initial knowledge structure, and we believe that the payoff is in being able to reuse generic architectures to design solutions for a family of problems. We need to identify such architectures and problem classes and use KASE for designing software systems for problems belonging to such problem classes.

The KASE system represents our initial attempt in building a prototype environment that can offer varying degrees of assistance to a software designer by employing diverse sources of knowledge. Our current work is focusing on extending the domain modeling representation to capture the dynamic behavior of a system by modeling states, transitions, events, and actions. We are also exploring the issue of design rationale capture and its reuse during redesign. KASE's current redesign capabilities were mentioned briefly in this paper. We are interested in extending these capabilities so that KASE can automatically incorporate certain changes in problem requirements into the design by using the design rationale.

Acknowledgements

The KASE system is a result of several people's work. We gratefully acknowledge the contributions made by Nelleke Aiello, Raymonde Guindon, Liam Peyton and Go Nakano who wrote most of the code for KASE.

References

Adelson, B. & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transaction on Software Engineering*, SE-11(11):1351 - 1360.

Bhansali, S. (1992). Generic software architecture based redesign. AAAI Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse, Stanford, CA.

Bhansali, S. & Nii, H. P. (1992). KASE: An integrated environment for software design. *2nd International Conference on Artificial Intelligence in Design*, Pittsburgh, PA.

Brown, H. D., Schoen, E., & Delagi, B. A.(1986). An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Department of Computer Science, Stanford University, Technical Report STAN-CS-86-1136.

Graves, H. (1991). Lockheed Environment for Automatic Programming. 6th Knowledge-Based Software Engineering Conference, Syracuse, NY: 78-89.

Guindon, R. (1990). Designing the Design Process: Exploiting Opportunistic Thoughts. *Human-Computer Interaction*, 5:305-344.

Guindon, R. (1992). Requirements and design of DesignVision, an object-oriented graphical interface to an intelligent software design assistant. *ACM Proceedings of CHI'92*, Monterrey, CA.

Guindon, R., Krasner, H., & Curtis, B. (Eds.). (1987). *Breakdowns And Processes During The Early Activities Of Software Design By Professionals*. Ablex Publishing Corp.

Nii, H. P., Aiello, N., Bhansali, S., Guindon, R., & Peyton, L. (1991). Knowledge Assisted Software Engineering (KASE): An introduction and status June 1991. Knowledge Systems Laboratory, Computer Science Department, Stanford University, Technical Report KSL-91-28.

Nii, P.(1989). Blackboard Systems. In A. Barr, P. Cohen, & E. Feigenbaum (Eds.), *Handbook of Artificial Intelligence*. New York, NY: Addison-Wesley.