Knowledge Systems Laboratory
Report No. KSL 93-60

August 1993

Software Synthesis using Generic Architectures

by

Sanjay Bhansali

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, California 94305

Software Synthesis using Generic Architectures

Sanjay Bhansali
Knowledge Systems Laboratory
Department of Computer Science, Stanford University
701 Welch Road, Building C
Palo Alto, CA 94304

Abstract

We describe a framework for synthesizing software systems based on abstracting software system designs and the design process. The result of such an abstraction process is a generic architecture and the process knowledge for customizing the architecture. The customization process knowledge is used to assist *a designer in customizing the architecture as opposed to completely automating the design of systems. We illustrate our approach using an implemented example of a generic tracking architecture which we have customized in two different domains. We describe how the designs produced using KASE compare to the original designs of the two systems, describe current work and plans for extending KASE to other application areas.*

1 Introduction

Synthesizing software systems by reusing previously developed components has long been a subject of considerable interest in software engineering. One of the most effective principles that has emerged for reusing software is *abstraction*. Abstraction consists of extracting the inherent, essential aspects of an artifact, while hiding its irrelevant or incidental properties. One of the ways in which abstraction fosters reuse is by providing a class of artifacts that can be *instantiated* or *customized* to produce several different artifact instances meeting different requirements. Procedural and data abstraction, information hiding, and parameterized programming are examples of some of the most notable application of the abstraction principle in software systems.

The abstraction principle has also been used as the basis for *automating* the construction of artifacts that would normally require a creative process. For example, Emycin (van Melle, 1980), an expert system shell was developed by abstracting the control structure of Mycin; abstracting out the process of building blackboard systems yielded AGE (Nii & Aiello, 1979). Commercially available expert systems shells and application generators are based on different mixtures of design and process abstractions. More recently, abstraction has been successfully used in algorithm synthesis, e.g., the KIDS system (Smith, 1990) contains abstractions of several different classes of algorithms in the form of algorithm theories which can be (semi-)

automatically instantiated to synthesize specialized algorithms for several different problem instances.

In the KASE (Knowledge Assisted Software Engineering) project (Bhansali & Nii, 1992a; Guindon, 1992) we are investigating the utility of abstracting *software system designs* and *the design process*. Designing software systems is a creative and ill-understood process. Successful software designs are created by a small group of designers; however, the process is rarely documented and the final design is typically not well documented. Consequently, it is difficult to understand and maintain the system, which in turn leads to poor reuse. Our approach to this problem consists of (1) identifying useful classes of software systems and the problems they solve, (2) abstracting the design of the system as a *generic architecture* for that class of problems, (3) formulating rules and constraints for customizing the architecture based on specific problem descriptions, and (4) providing a computational environment that enables designers to construct specific systems semi-automatically by customizing the generic architecture. Such an approach allows us to reuse the architecture for multiple applications within the class, capture the process of software design which could be used to maintain the system (Bhansali, 1992) or be reused for multiple designs, and ultimately, learn algorithmic descriptions of the design process (Garg & Bhansali, 1992).

A guiding theme in our research is to provide a set of software tools that support the way humans design. Thus, our goal is not to create a fully, automated software synthesis system, but rather to provide a mixed-initiative system in which the design task is divided between a human designer and the system. Typically, KASE provides design alternatives and default suggestions for architectural parameters, explanations for its suggestions, dependency maintenance between different design decisions, and consistency checking. The human designer determines the order in which the various design actions are initiated and makes the final choice for each design decision, which may or may not be based on the suggestions offered by KASE.

Our approach may be characterized as a semi-formal approach. It is not completely formal where the semantics of a problem specification and architectural descriptions are contained entirely within a set of mathematical equations. Nor is it completely informal where the name of a symbol carries all the information for a human as in, e.g., systems like IBIS (Conklin & Begeman, 1989) and hypertext. Our approach relies instead on keywords and commonly accepted domain-specific ontology which are not formally defined. However, there are explicitly represented constraints and rules that provide some semantics to the symbols. We were motivated in adopting this approach because we wanted to create a practical system that could be used by software designers who are not well-versed in formal, mathematical notation; at the same time we wanted a machine to be able to reason with the representation, draw useful inferences, and provide intelligent assistance to designers in customizing generic architectures.

We have used KASE to design systems in three different application areas (Bhansali, 1993; Bhansali & Nii, 1992b). In this paper we describe our approach and experience in synthesizing software designs in one such application. The application is that of tracking a set of moving objects (e.g. aircraft) based on an analysis of signals emitted by them. We have designed two different systems in this application area. The two systems had been originally designed several years ago by two different teams of designers. What we have shown in KASE is a rational reconstruction of the design of these two systems by reusing a single generic architecture and the same design rules. In section 7 we compare the designs that were produced by KASE with the original designs of the two systems. The comparison shows that the designs that were produced by KASE were more systematic, more comprehensible, and less likely to have errors due to omission. On the other hand they were not as efficient as the original designs and required some amount of application-specific optimization.

1.1 Framework for Architecture-based Software Design

Figure 1 shows an overview of the KASE system. The shadowed boxes represent knowledge components that are part of KASE. Figure 2 gives an overview of the process of synthesizing systems using KASE.

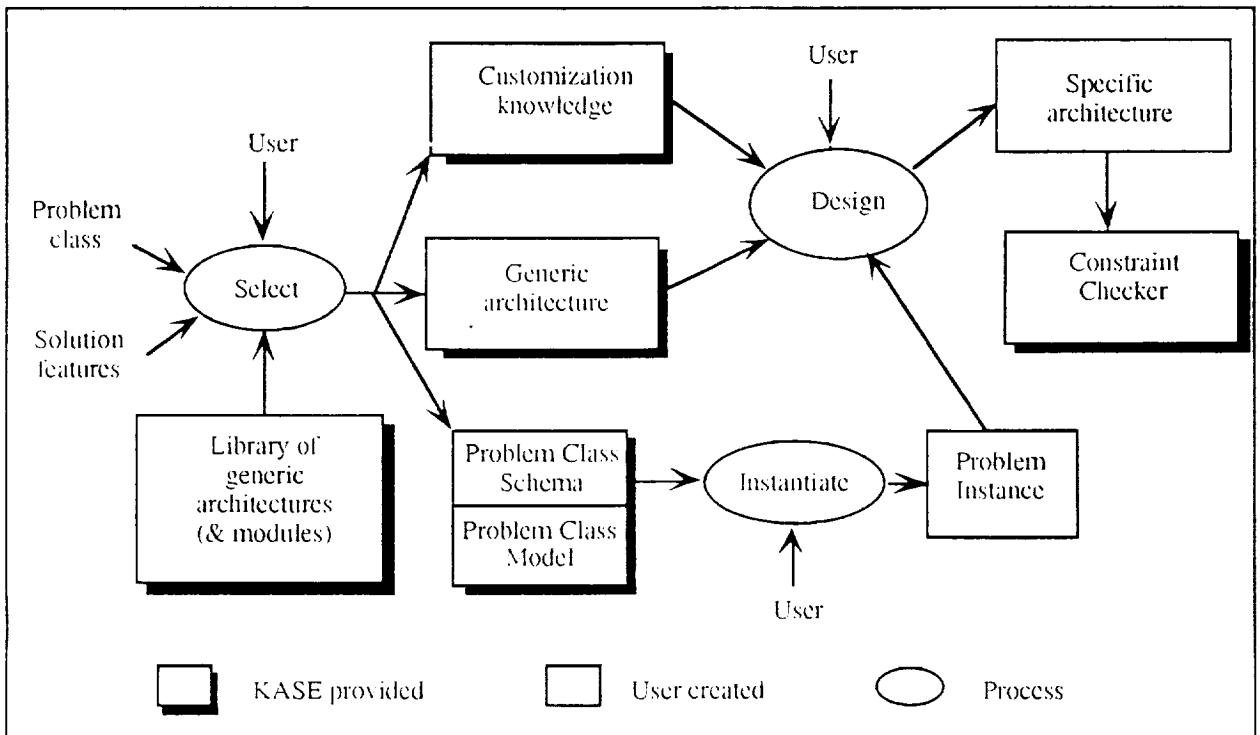


Figure 1. Overview of software design in KASE

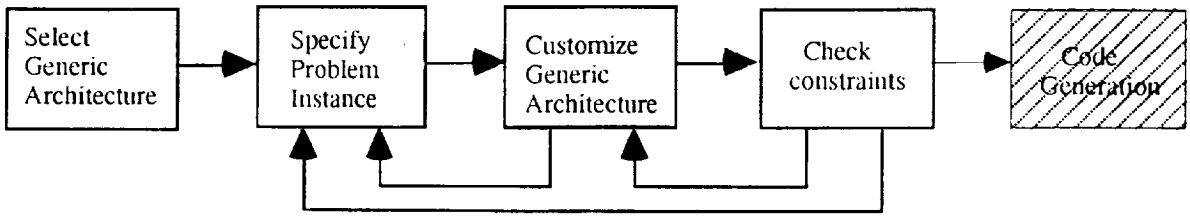


Figure 2. Process model for synthesizing software systems using KASE. The shaded box denotes process not described in this paper.

A designer initiates the design process by first *selecting* a *generic architecture* from a library based on the *problem class* for his particular problem and the desired *solution features* (Section 2). Associated with the generic architecture is a specification for the problem class called a *problem-class schema* as well as a *problem-class model*. An individual problem is specified by instantiating the problem class schema; the problem-class model contains the vocabulary of terms that help in the instantiation (Section 3). Also associated with the generic architecture is *customization knowledge* which contains knowledge for customizing the generic architecture and is the basis of KASE's intelligent support (Section 4). Finally, KASE has a constraint checker that is used to check for the consistency of the design with respect to certain architecture-specific constraints (Section 5).

Figure 3 illustrates the general relationship between generic architectures, problem-class models, and customization knowledge. It can be seen that a generic architecture may be used to solve different problems belonging to different problem classes; likewise a problem may be solved using different generic architectures. For example, Generic Architecture 1 may be used to solve all instances of Problem Class 1 as well as all instances of Problem Class 2, and instances of Problem Class 2 may be solved using either Generic Architecture 1 or Generic Architecture 2. The customization knowledge is the crucial link between a generic architecture and a problem class; it contains rules that determine how the parameters of a generic architecture must be instantiated in order to solve problem instances.

In our current work we have shown how a single generic architecture can be customized to solve two different problems that are instances of a problem class by reusing a common customization knowledge. Investigating how a single generic architecture can be used to solve problems belonging to different problem classes, and how a single problem instance can be solved on different generic architectures are topics that we have left for future work.

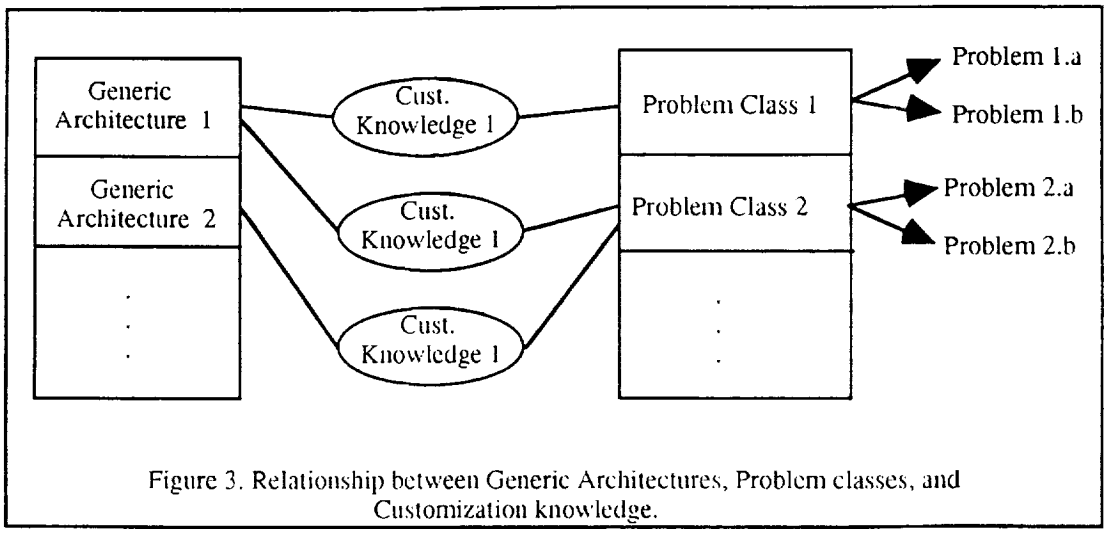


Figure 3. Relationship between Generic Architectures, Problem classes, and Customization knowledge.

Subsequent sections describe each of the processes in Figure 2 - selection of an architecture, problem specification, customization, and consistency checking - in detail. At this point it might be useful to look at an example from a familiar domain to help ground the concepts and terminology in KASE. The example is for illustration purposes only and has not been implemented in KASE.

1.2 Example: Compiler synthesis

Suppose we were to use KASE to help software designers in designing compilers. The *problem class* can be described as follows: *Given the syntactic and semantic specification of a source language and a target language, design a system that takes as input a string, and, provided the string is syntactically valid in the source language, produces an (semantically) equivalent string in the target language.*

The *problem class schema* consists of a set of *roles* representing the parameters of the problem and *constraints* on the values of the roles. For this example the problem schema would have as roles the *source language syntax* constrained to be a context-free grammar and the *source language semantics* constrained to be specified in, say, an operational form. The *problem-class model* would have terms like grammar, context-free grammars, regular grammars, productions, start symbol, terminals, non-terminals, and other auxiliary concepts that a user needs to know in order to specify problems as an instance of a problem class (i.e. fill the problem-schema roles with values). A single *problem instance* would then consist of a particular grammar (a set of productions, a start symbol, the set of terminals, and the set of non-terminals), a specific instruction set for a target language, the meaning of each syntactically valid string that can be generated from the start symbol, and so on.

A *generic architecture* for this problem class might be as shown in Figure 4 (adapted from (Aho & Ullman, 1977)). The architecture is an organization of generic modules that implement

the main phases of a compilation process. The *solution features* associated with this generic architecture might be, e.g., that this is a single-pass compiler, or that it requires time and space that is O(n) where n is the length of the input string. The customization process would consist of determining the detailed algorithms and data structures for implementing each of the generic modules. The customization knowledge would consist of rules that specify how properties of an individual problem specification suggest/constrain the implementation choices for the generic modules. An example of such a rule would be: *If no production right side in the source grammar is ε or has two adjacent non-terminals, then an operator-precedence algorithm may be used as the parsing algorithm in the parser module.* Finally, the constraint-checker would have certain constraints that need to be satisfied by the design of any compiler. For example, *if an LR parsing*

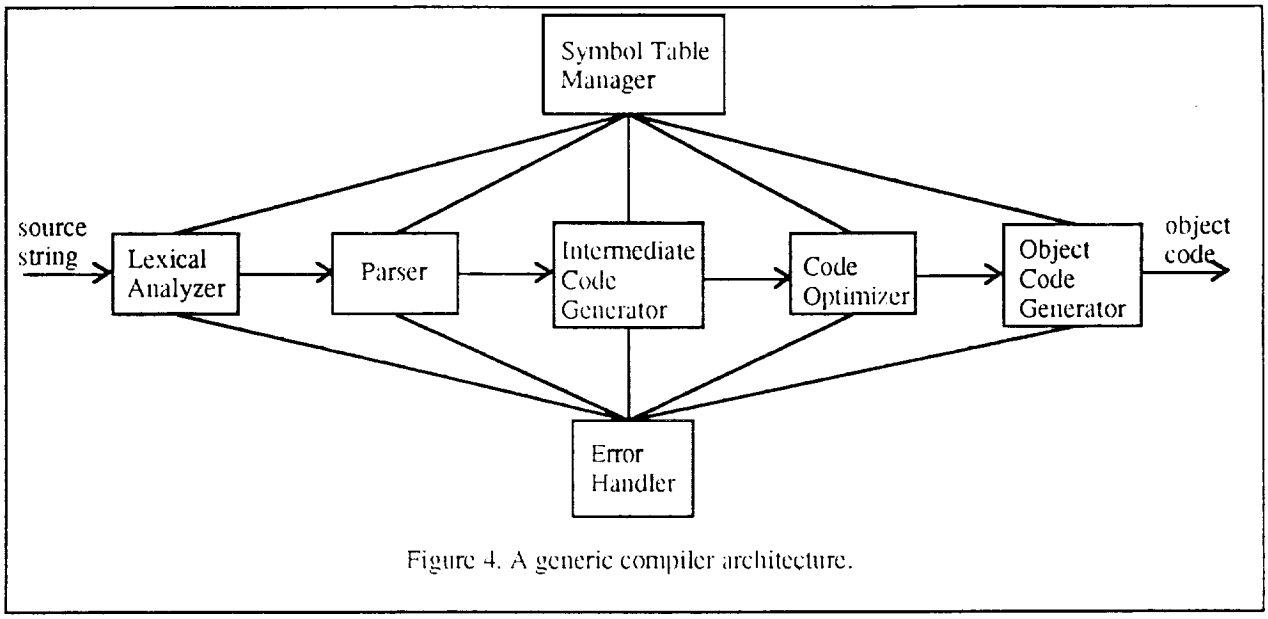


Figure 4. A generic compiler architecture.

algorithm is being used then the action entry of the LR parsing table should be unique (i.e. there are no conflicts).

The above is a pedagogical example given in order to elucidate the main concepts in KASE in terms of a well-known and familiar example. For the rest of the paper we will use the tracking application domain to illustrate in detail the design process in KASE. Specifically, we will consider the following problem:

Ship and Submarine Tracking Problem: There is a region of ocean in which ships and submarines are moving. As they move they emit noise which can be detected by sonar sensors. Often the received signal is distorted by background noise and other objects in the environment. The problem is to design a system to analyze the properties of all the signals received by sensors

located at collection sites, determine the identity, location, heading, and other characteristics of the ships and submarines, and report them periodically (see Figure 5).

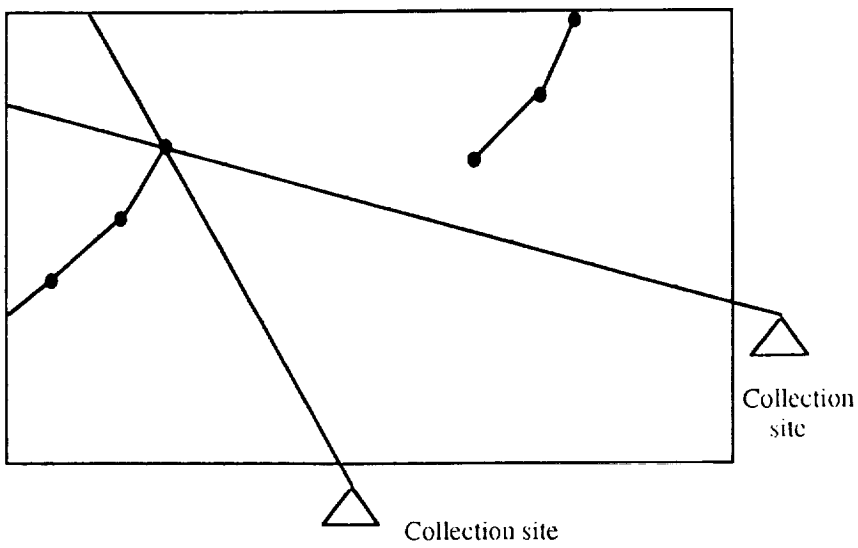


Figure 5. Tracking ships and submarines. The lines represent tracks and the circles represent positions of ships and submarines at different times.

2. Architecture Selection

The process of designing a system in KASE begins by a designer selecting an appropriate generic architecture from the library. The library of architectures is indexed to help in the selection process. The indexing scheme is based on two main hypothesis. Broadly speaking, the hypotheses state that generic architectures are a cross product of problem-classes and the high-level design decisions for solving the corresponding problem instances.

2.1 Problem Class Hypothesis

The first hypothesis is that *architectures are designed to solve a class of problems that share certain features*. For example there may be an architecture that is designed to analyze signals in a batch; this architecture would be different from another architecture that performs analysis of continuous signals; these two architectures would be radically different from, say, a real-time interactive system that is governed by strict timing constraints and user interactions. According to the problem class hypothesis, we can use problem classes as one component of an index to generic architectures. In KASE, when a generic architecture is created, an annotation is attached to the architecture that describes the class of problems which can be solved using the architecture. Subsequently, during architecture selection, KASE presents a list of all architectures

and their annotated problem class descriptions. From this description a designer can determine to which problem class his or her problem belongs and use that to select a generic architecture.

A problem class description is obtained by abstracting the common features from a class of problems. This description would depend on how general the architecture is. For example, a pipelined architecture is a very general architecture and the associated problem class would be very general. On the other hand, a pipelined architecture for a compiler is quite specific with the number of components and the interfaces of each component clearly defined; consequently, the associated problem class - compiling a language of a certain type - is quite specific.

As a concrete example in KASE, there is a generic architecture for a problem class called *tracking* which is described as: *Track moving objects based on signals received from the objects and infer properties of the moving objects.* (Section 3.1 shows how this problem class is represented in KASE.) Thus, in addition to the Ship and Submarine tracking problem, this architecture could be used, say, to track aircraft based on their radar emissions, or any other moving objects that emit a detectable signal.

We currently do not address the issue of how specific problem classes can be identified and associated with generic architectures. There has been some related work (Shaw, 1991) in identifying common architectural idioms and how features of a given problem make a particular architectural choice appropriate or inappropriate. However, this is an open problem requiring further research.

2.2 Solution Features

A problem class by itself is not sufficient to determine an appropriate architecture for a problem because there may be several, quite different designs for a particular problem depending upon how various design issues and trade-offs involved in decomposing and solving a problem are resolved. Our second hypothesis governing the selection of architectures is that *an architecture embodies a set of high level strategic decisions on how to decompose and solve a problem.*

In general, these strategic decisions depend on the problem class. For example, for the tracking problem one of these strategic decisions would be whether to use concurrency or not. For some other problem class a strategic decision might be whether the system functionality should be decomposed into a set of horizontal layers or whether to use weakly coupled vertical partitions. For yet another problem class the degree of accuracy required in the solution may be the overriding factor in selecting an architecture. The way a strategic decision is resolved depends on the requirements of a problem. The collection of strategic decisions associated with a generic architecture is called the *solution features* - they characterize the solution to a problem.

The size of the solution features set depends on the degree of abstraction of an architecture. For a completely instantiated architecture this size would be maximal and correspond to the set of *all* design decisions made in going from the requirements of a problem to the final design. For a generic architecture, the solution features would be some subset of the solution features of the fully instantiated architecture. The solution features are represented as attributes associated with a generic architecture. For example, the solution features associated with one generic architecture for tracking are as shown in Figure 6.

Signal-processing-strategy[*symbolic,statistical*]: symbolic
Processing-platform[*uni-processor, multi-processor*] : uni-processor
Processing-mode[*incremental, batch*] : incremental
Processing-knowledge[*algorithmic, heuristic*] : heuristic

Figure 6. Solution features associated with a generic tracking architecture. The terms in italics represent choices for the features and the term on the right of the colon represents the feature that is embedded in one particular generic architecture.

The solution features and the various alternatives choices for them are not defined formally and are not used by KASE to do any automated inferences. They simply serve as keywords for indexing the generic architecture library. However, once the various solution features have been identified, they may be used as a basis for designing other generic architectures by taking different combinations of the solution features, for example a generic architecture for tracking that uses a multiprocessor and processes input signals in a batch.

For each generic architecture that is selected for a problem class, KASE presents to the user the solution features embedded in the architecture and based on the requirements of a problem the user selects one of the architectures.

3 Problem Specification

Having identified generic architectures with problem classes, we can create and store generic descriptions for the problem classes along with the architectures. In addition, we can create a model for the problem-class which contains the vocabulary of concepts relevant for describing problems belonging to the problem class. Individual problem instances are then specified by instantiating the generic problem-class description using the problem-class model.

3.1 Problem Class Description

We represent a class of problems as a problem *schema*. A problem schema consists of a set of *roles*, which represent the parameters of a problem, and *constraints* on the values of the roles. Instantiating these roles with specific values produces a problem specification instance. Figure 7 shows the problem schema for the tracking problem class.

objects-to-be-tracked: *trackable-objects* =

relations-to-be-tracked: *trackable-relations* =

attributes-to-be-tracked: *attributes* =

tracked-object-behavior: *state-diagrams* =

tracked-relation-behavior: *state-diagrams* =

tracking-operations: *operations* =

input-signal: *signal* =

signal-collectors: *collection-sites* =

Figure 7. Tracking Problem Class schema.

The terms to the right of the colon denote type restrictions on the problem parameters. In general, there may be other constraints in addition to the type constraints on the values of the parameters. For example, in the tracking problem class there are constraints that check that the user specifies at least one operation that takes as input a signal record and computes attributes of an object that is an instance of objects-to-be-tracked. (It is possible to implement this constraint because operations are specified declaratively in terms of inputs, outputs, preconditions, and postconditions). Other examples of constraints implemented for this problem class are given later. To specify individual problem instances, a user has to provide values for each of the roles in the problem class schema such that the corresponding role constraints are satisfied.

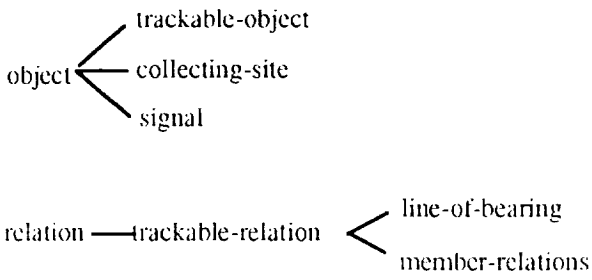
3.2 Problem Class Model

The types that appear in the problem class schema are part of the *problem-class model*. In general, a problem class model contains the basic concepts or vocabulary necessary for modeling a class of problems. This includes classes of objects, common attributes of the objects, generic relations between them, and the specifications of generic operations. For capturing the dynamic behavior of the objects, the model might contain the minimal set of states for various objects, transitions between these states, and events triggering these transitions. Note that it is not critical for the problem-class model to be complete (e.g., contain all relevant classes of objects needed to model a particular application) since our objective is not to provide complete automation for the software design. However, the assistance that KASE can provide to a designer is based on its customization knowledge (Section 4) which in turn depends on how complete the problem class model is. Therefore, a relatively complete problem-class model enhances the usefulness of KASE.

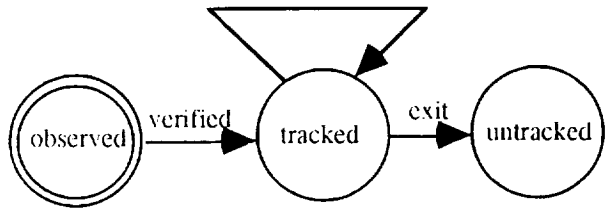
In KASE, problem-class models are created using an object-oriented modeling methodology (Rumbaugh,Blaha,Premerlani,Eddy, & Lorensen, 1991). The model description begins by first

defining the types used in the problem class schema. Objects, relations, attributes, operations, states, events, transitions, and a set of primitive data types (that are provided by KEE[1], the underlying object-oriented environment) constitute the set of basic available types. All other types are defined as subtypes of this basic set. In object-oriented modeling a model consists of three parts: a *static* model, a *dynamic* or *behavior* model, and a *functional* model. The static model consists of the objects, relations, attributes, and operations on the objects and relations. The behavior model shows the temporal relationships between objects and relations in terms of states, events, and transitions between the different states. This is shown using a state diagram. Finally, the functional model specifies the meaning of operations specified in the object model. The operations are typically specified in terms of the inputs, outputs, preconditions, and postconditions, and/or an equation relating the output to the input.

Figure 8 shows fragments of the static, behavior, and functional model for the tracking problem class. For each object and relation in the static model, common attributes, constraints on the type of attributes, and common operations are also specified. The behavior model is shown as a state-diagram in KASE. KASE contains a graphical user interface that allows users to create and edit such diagrams graphically (Bhansali, 1993).



(a) Static model



(b) Behavior model: state transition diagram for objects-to-be-tracked. The labels on arcs represent events.

```
(DEF-OPERATION compute-speed-from-2-positions
  :input ((?p1 position) (?t1 time) (?p2 position) (?t2 time))
  :output ((?s real))
  :body assign[?s, vector-norm[vector-subtract[?p2 ?p1]]/(?t2 - ?t1)]
  )
```

(a) Functional model: definition of an operation to compute an attribute of a trackable-object.

Figure 8. Fragments of the static, behavior, and functional model for the tracking problem class.

[1] KEE is a registered trademark product of Intellicorp Inc.

3.3 Problem Instance Specification

The problem class schema and the problem-class model are used to drive the acquisition of specifications for individual problem instances. Before a problem instance is specified, a user extends the problem-class model by introducing problem-specific terms. For example, the user may create new classes of objects and relations as specializations of the objects and relations in the problem-class model. A problem class model helps in this process in three ways:

1) It provides an organizational structure for the problem-specific knowledge and communicates to a user the kinds of knowledge needed and how to represent them. For example, consider Figure 9 which shows the extension of the tracking problem-class model for two different problem instances. It can be seen that many different kinds of objects like an emitter (an object that emits radar signals), a cluster (a set of aircraft sharing certain properties), a line (which represents a signal of a particular frequency), and a harmonic (a set of related lines) are all classified as a subclass of trackable-object. This is because the existence of each of these entities can be inferred from the information available in a signal. Without the organizational framework provided by the problem-class model, these objects might have been classified differently by different users. As we will see later, the organization of the problem-specific knowledge is important in determining the support that KASE can provide to a user during customization.

Note, that we are making a crucial assumption here: the meaning of the various concepts that constitute the problem-class model are shared by the problem-class modeler and the problem specifier. The use of mnemonic names, textual annotation, and explicitly represented constraints facilitate the sharing to some extent. However, general techniques for communicating such ontological commitments is a research topic beyond the scope of our current work (Gruber, 1991).

2) The problem-class model also provides default types and values for certain attributes and default definitions for certain operations. For example, the co-ordinate system for representing the position of objects, and the definition of an operation that computes the average speed of an object given its position at two different times, can be inherited directly from the problem-class model. This reduces the amount of effort expended in specifying new problem instances.

3) Finally, certain constraints can be built-in to validate a problem specification for consistency. For example, for the tracking problem there is a constraint to ensure that for at least one tracked object, the user defines at least one operation that takes as input a signal and creates an instance of the tracked object. (Note, that in general it is not necessary that *each* trackable object instance be inferred directly from the signal - some of them may be inferred indirectly through the existence of other objects). Similarly, there are constraints that check that the states used in defining operations have been defined in the state transition diagram, the attributes referenced in the operation definitions have been defined in the static model, and so on.

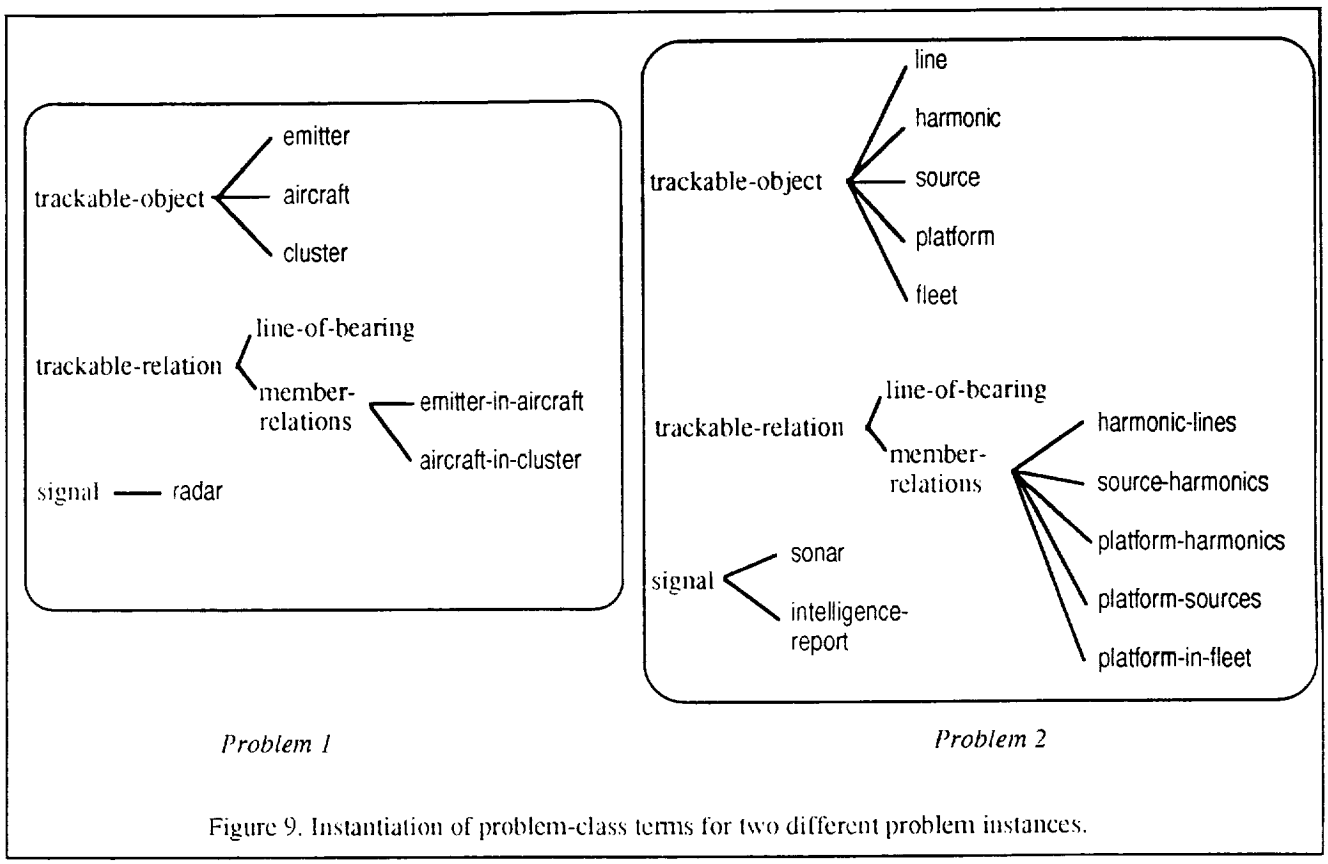


Figure 9. Instantiation of problem-class terms for two different problem instances.

Figure 9 showed the extension of the tracking problem-class model for two different problems – tracking aircraft based on processed radar signals (Brown,Schoen, & Delagi, 1986) and tracking ships based on processed sonar data and intelligence reports. The attributes of a typical domain object and the operations associated with the object are shown in Figure 10.

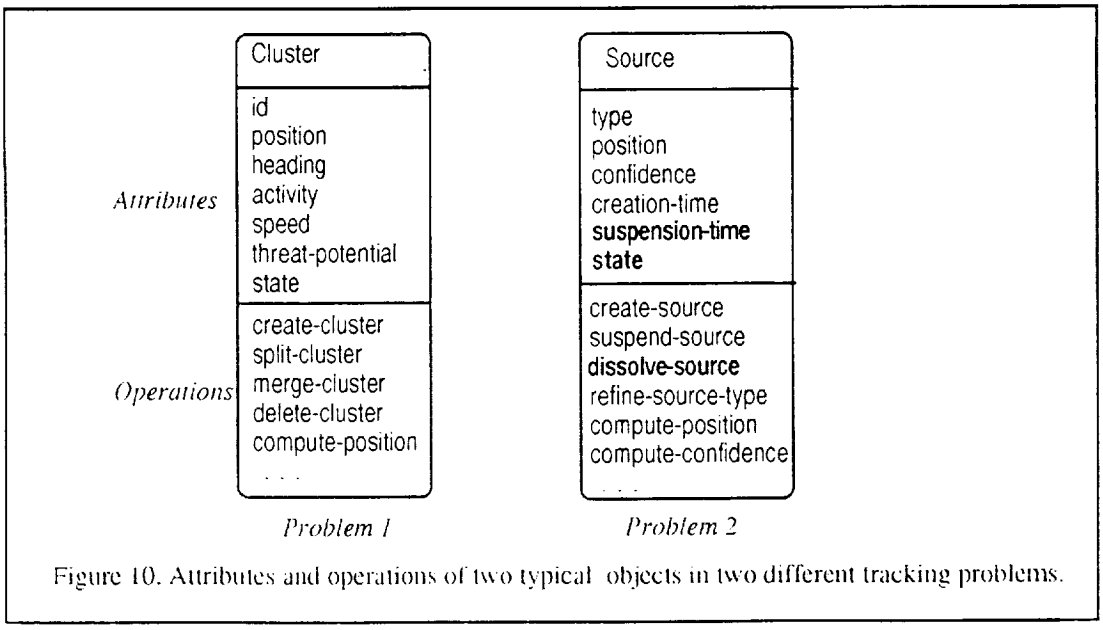
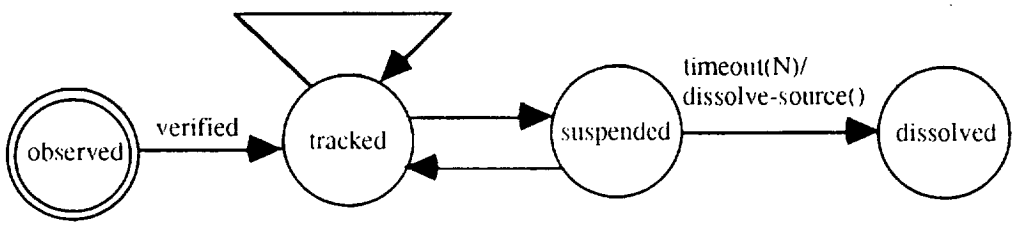


Figure 10. Attributes and operations of two typical objects in two different tracking problems.

An operation is specified in terms of the inputs, outputs, precondition, postcondition, and (optionally) an expression in a high-level language. The generic state transition diagrams associated with trackable-object is intended to provide a starting point to a user in specifying the operations. Typically, the user would copy the generic state transition diagram associated with an object and then modify it using the graphical tools provided by KASE. Once the state transition diagram is customized, the designer would fill in the definitions of each operation. Figure 11 shows the customization of a state-transition diagram for the source object and the definition of an operation on it.



(a) Instantiation of a generic state transition diagram for an object called source *(see Figures 8 & 9)*

```
(DEF-OPERATION dissolve-source
:comment "If a source is suspended for more than N time-units
          then remove it from further consideration."
:input ((?s source) (?t time))
:precondition (AND (= (state ?s)-SUSPENDED)
                   (> (- ?t (suspension-time ?s)) N))
:body ()
:postcondition  (= (state ?s) DISSOLVED)
```

(b) Definition of the dissolve-source operation appearing in the above diagram. suspension-time *and* state *are attributes of* source *(see Figure 10).*

Figure 11. Specifying a problem instance: behavioral and functional aspects

4 Customizing a Generic Architecture

Just as a problem class is an abstraction of a set of problem instances, a generic architecture is an abstraction of the solutions for a set of problems. It is obtained by abstracting the common features from the solutions of a set of problems. Figure 12 shows a generic architecture for the tracking problem. This architecture is based on the blackboard model and its solution features are shown in Figure 6. These features of the architecture satisfy certain problem requirements, e.g., computational cost - an architecture based on conventional statistical processing instead of symbolic manipulation of the data could be computationally too expensive. (Nii,Feigenbaum,Anton, & Rockmore, 1982) gives a more detailed description of the rationale for using a blackboard-based architecture for these kinds of problems.

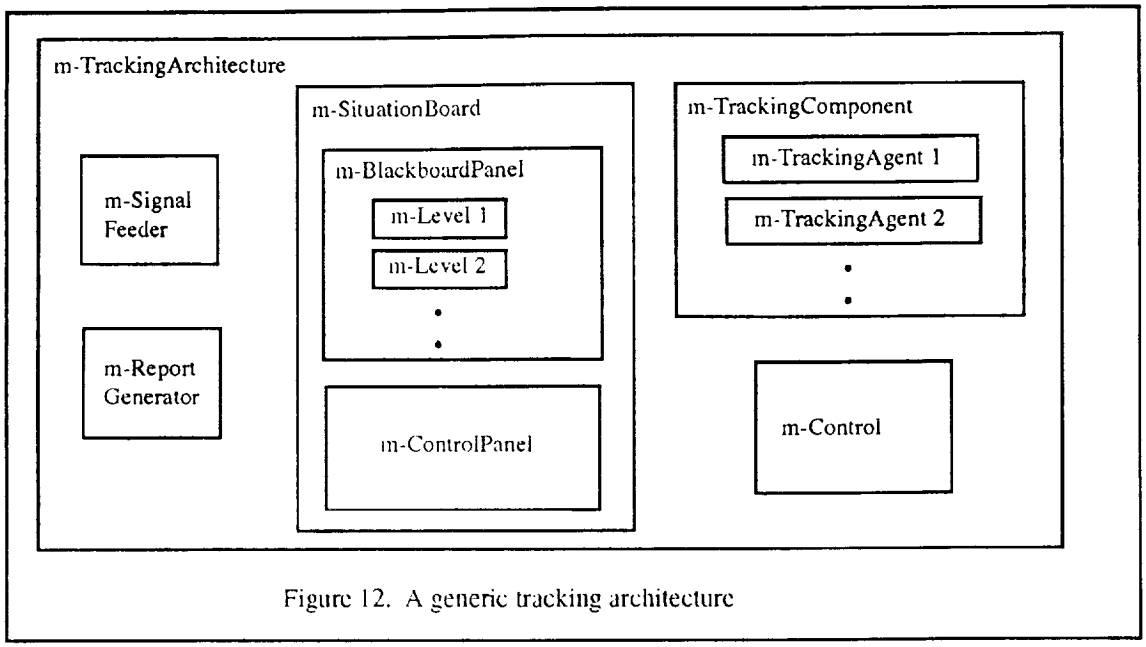


Figure 12. A generic tracking architecture

The architecture consists of five major submodules (a module is defined shortly). The m-Signal-Feeder and m-Report-Generator modules represent the system's interface to the external world and contain routines to read the signals provided by collection sites and routines to generate periodic reports respectively. m-SituationBoard is used to represent the state of the various tracked objects and relations as well as certain control information. m-TrackingComponent contains submodules called m-trackingAgents that compute the values of the data represented in m-SituationBoard. m-Control contains routines that monitor the activity on the m-SituationBoard and decides what action to take next. (This is a simplified description of a blackboard-based architecture. For more details see (Jagannathan,Dodhiawala, & Baum, 1989; Nii, 1989)).

4.1 Generic Architecture

We define a *module* as a packaging of procedures and/or data in a logical unit. In KASE, a module is represented as an object with a set of attributes. Figure 13 shows the minimal set of attributes for each module. Attributes that are preceded by an * are derived attributes whose values are computed from the primitive attributes (e.g., the input to a module is simply the set of variables that form arguments to procedures provided by the module and the results of procedures required by the module). For each attribute, there is a type and cardinality constraint on the values that can be used to instantiate it. This is indicated by the keywords *:valueclass* and *:cardinality*, respectively. The cardinality is specified as a range (*min - max*) with *?* indicating that there is no restriction on the maximum cardinality. Thus, for example, the submodule slot of a module can be either nil or be instantiated to a set of any number of modules, whereas the supermodule slot can be at most a single module.

MODULE

submodules *modules contained within this module*
:valueclass module
:cardinality (0-?)

provides *procedures provided by this module*
:valueclass procedure
:cardinality (1-?)

requires *procedures required by this module*
:valueclass procedure
:cardinality (0-?)

has-locally *local procedures*
:valueclass procedure
:cardinality (0-?)

has-access-to *modules which can provide procedures to this module*
:valueclass module
:cardinality (0-?)

***supermodule** *module that contains this module (inverse of* submodule*)*
:valueclass module
:cardinality (0-1)

***inputs** *data flow into the module*
:valueclass any
:cardinality (0-?)

***outputs** *data flow out of the module*
:valueclass any
:cardinality (0-?)

***calls** *modules called by procedures within this module*
:valueclass module
:cardinality (0-?)

annotation an English description of the functionality of this module
:valueclass string

Figure 13. Minimal internal representation of a module.

Our representation of a module is similar to the traditional notion of a module in the literature (Prieto-Diaz & Neighbors, 1986), except that we do not have a slot to represent declarations for data structures. Data structures are represented using the notion of data encapsulation, i.e. as an instance of an abstract data type. Thus, instead of providing a data structure that can be accessed or modified by external routines, a module simply provides a set of operations that can be performed on it. For example, consider the module m-signal-feeder which is implemented as a file containing a sequence of records, where each record corresponds to a signal obtained from a collection site. Such a module is represented as:

module m-signal-feeder
provides open-signal-file, close-signal-file, read-next-record.
requires nil
...

Since there is no write operation, other procedures can only read records from this file. Other examples of modules in the tracking architecture are shown below.

module m-Control

submodules nil

provides p-simple-control
:valueclass ONE.OF (p-simple-control,
p-hybrid-control)

requires get-TrackingAgent-triggers,
get-BB-levels,
get-BB-level-instances,
get-posted-events, . . .

has-access-to m-TrackingComponent,
m-SituationBoard

has-locally determine-executable-agents,
determine-BB-nodes,
schedule-operations,
execute-schedule

module m-TrackingComponent

submodules __
:valueclass m-TrackingAgent
:cardinality (1-?)

provides get-TrackingAgent-triggers,
execute-Agent

requires get-BB-level-state,
put-BB-level-state

has-access-to m-SituationBoard

A module interface is defined in terms of the procedures (or operations) that it provides to other modules, and the procedures that it requires from other modules. The other attributes are used to constrain the way a system is structured and the way modules communicate with each other. For example, a module may only use procedures provided by its submodules or a module that it has access to.

A *generic module* is an abstraction of a set of modules obtained by viewing some of the attributes of a module as parameters. Although, technically each of the module attribute may be considered as a parameter, we have found that the two most useful one are the *submodules* and the *provides* attribute of a module. For example, in the tracking architecture the *m-TrackingComponent* module is a generic module in which the *submodules* attribute is a parameter. A specific instance of *m-TrackingComponent* is obtained by instantiating this attribute with a specific set of submodules. Similarly, the *m-Control* module shown above is a generic module with the *provides* attribute being a parameter. There are constraints which determine how the parameters of a generic architecture may be instantiated. One of the constraints is a type (or valueclass) constraint. For example, for the *m-TrackingComponent* the submodules are constrained to be of type *m-Tracking-Agent* which is another generic module, and for the *m-Control* module, the provides slot is constrained to be either *p-simple-control* or *p-hybrid-control* - which are generic procedures.

4.2 Generic Procedures

Analogous to modules, there is a notion of generic procedures in KASE. A procedure is represented in KASE as shown in Figure 14. A generic procedure is obtained by treating the

inputs, outputs, or body of the procedure as a parameter. The valueclass for the body attribute of a procedure is a *program-schema* or a template similar to the notion of a *cliché* in Programmer's Apprentice(Waters, 1985); a specific value for the *body* is a refinement of the program-schema such that the pre- and post-conditions of the procedure are satisfied. Currently, KASE does not support the verification of programs, and hence the pre-conditions and post-conditions are not being directly used for supporting code synthesis. However, section 6.5 describes how certain design rules use information provided by the pre- and post-conditions to suggest parameter values.

PROCEDURE
- **inputs**
  - :valueclass (id, datatype)
  - :cardinality (0-?)
- **output**
  - :valueclass (id, datatype)
  - :cardinality (0-1)
- **precondition**
  - :valueclass logic-expression
  - :cardinality (0-1)
- **postcondition**
  - :valueclass logic-expression
  - :cardinality (0-1)
- **body**
  - :valueclass program-schema
  - :cardinality (1-1)
- **annotation**
  - :valueclass string
  - :cardinality (1-1)

Figure 14. Representation of a procedure in KASE

As an example, consider the program schema[2] associated with a generic procedure called p-simple-control:

```
procedure p-simple-control
    ...
    body =
      loop[{ par[assign[agents, determine-executable-agents[]],
                 assign[nodes, determine-BB-nodes[]]],
             assign[schedule,determine-schedule[agents,nodes]
             call[execute-schedule[schedule]]},
            termination-condition[]]
```

[2] The program schemas are written in a simple language that is derived from the syntax of Mathematica. They can be easily converted to code in some executable language. Currently there exists a translator that converts programs written in this language into Fortran.

The above procedure determines a set of tracking-agents and nodes (instances of objects and relations represented in the SituationBoard), creates a schedule for executing the tracking-agents on the relevant nodes, and then executes the schedule. The construct *par[statement$_1$, statement$_2$]* means that the order in which the two statements *statement$_1$* and *statement$_2$* are executed is unspecified. During customization, one of the customization steps would be to determine the order of execution of these two statements. Depending on which statement is performed first, we get two quite different implementations of the control algorithm. (In blackboard terminology, this is referred to as determining the *focus of attention.*) Each of the operations in the program schema (e.g., *determine-executable-agents, schedule-operations*) may be either a subroutine from a code library or may be another program schema.

The above is an example of a very specific program schema. At another extreme, we have a very general program schema like the following:

```
procedure Tracking-operation-schema
    input (istate : BB-level-state)
    output (ostate : BB-level-state)
    precondition true
    postcondition true
    body
      valueclass: program-schema
```

This is a general class of procedures that take as input an instance of type BB-level-state (which represents the state of all objects and relations represented in the m-Level modules of m-BlackboardPanel)) and produces a new BB-level-state. The only restriction is that the operation terminates. As described shortly, KASE contains customization knowledge, in the form of rules, that suggest to a designer the set of all useful instances of the *Tracking-operation-schema* based on the customization of m-Blackboard-Panel.

With the above definition of modules, a *generic architecture* can now be defined simply as the top-level generic module in a system. Thus, a generic architecture is defined compositionally in terms of its constituent (generic) submodules and procedures and the constraints on those submodules and procedures. The process of *customization* is then defined as the process of instantiating the parameters of the various generic modules and processes comprising a generic architecture.

4.3 Customization Knowledge

KASE provides active support to a user in customizing an architecture by providing a list of customization actions that need to be performed for each generic component (module or procedure), suggesting ways for doing the customization, and providing rationales for its

suggestions. The knowledge for providing this support is represented as *customization knowledge* for each generic architecture. This knowledge has to be acquired from experienced designers who have designed systems within the scope of a generic architecture. Techniques that can help designers in acquiring this knowledge are the subject of active research within the knowledge acquisition community but a discussion of these techniques is beyond the scope of this paper.

The customization knowledge associated with a generic architecture is essentially a set of rules or methods that can assist a user in finding appropriate values for the parameters of the generic modules and procedures in the architecture. In KASE, the customization knowledge is packaged as a set of *customization commands* which are associated with a generic component. A customization command is represented as shown in Figure 15.

The *suggestion-generator* is a pointer to a method (a lisp function) or a set of rules (written in KEE's rule language) which generates suggestions for instantiating the value of a parameter. During customization, the suggestion-generator is invoked to obtain a list of alternatives for instantiating a parameter value. The user may then select one of the suggested values for instantiating the parameter.

The *instantiating-method* attribute points to a method used to perform the customization. The method takes as input the selected parameter value. In most cases, it simply updates the value of the parameter and marks the parameter as being customized. In the case of procedure parameters, the instantiating method may also invoke a set of transformation rules to refine the procedure body.

CUSTOMIZATION COMMAND

generic-unit	*; pointer to generic unit with which this customization is associated*
:valueclass ONE.OF (module procedure)	
parameter	*; name of parameter*
:valueclass _	*; can be name of any attribute of the generic unit*
suggestion-generator	*; see text*
:valueclass lisp-function OR rule-set	
instantiating-method	*; see text*
:valueclass lisp-function	
depends-on	*; see text*
:valueclass _	*; any customization parameter or attribute of a problem-specific unit*
rationale	*; see text*
:valueclass string	

Figure 15. Representation of customization knowledge in KASE

The *depends-on* attribute contains a list of other customization parameters or objects from the problem-specification which are used in the methods for generating suggestions. In general, the value used to instantiate a parameter depends on the values of these parameters and objects. This information is used by KASE to restructure a chronological sequence of design steps into a dependency graph. If at some point in the design, a designer wishes to retract a certain design step or change the problem requirements, KASE uses the dependency graph to identify all and only those design steps that are potentially affected by the retraction. It then retracts all the affected design steps while retaining the effects of those design steps that are not affected by the change.

The *rationale* slot is provided so that a user may add his or her own comments on why a parameter value was instantiated to a particular value (if the user does not choose one of the KASE-suggested values). This is currently in the form of uninterpreted English text and is meant to serve as a design documentation.

An example of a customization command is the following:

customization-command Suggest-BB-levels

generic-unit m-BlackboardPanel

parameter submodules

suggestion-generator Blackboard-level-rules

instantiating-method submodule-instantiating-method

depends-on ((instances objects-to-be-tracked) (instances relations-to-be-tracked) ...)

This customization command can be used to determine the set of submodules of the module m-BlackboardPanel. Each of these submodules must be an instance of the generic module m-Level. The customization method is implemented as a set of rules called Blackboard-level-rules. An example of one such (paraphrased) rule is:

If X is a subtype of objects-to-be-tracked and attributes of X are to be reported

then there should be a submodule of m-BlackboardPanel of type m-Level that manipulates objects of type X .

The outcome of this customization command does not depend on any other parameter of the generic architecture but it depends on some problem class entities like objects-to-be-tracked and relations-to-be-tracked. The result of the customization affects the customization of another parameter of the generic architecture - the submodules of m-TrackingComponent, which would therefore have the above parameter in its *depends-on* slot. If at some point the set of objects that

need to be tracked changes, KASE can automatically undo the customization of these two (and other dependent) parameters.

Note that the customization knowledge depends on the generic architecture and the formulation of the problem-class model for the corresponding problem-class. However, it is independent of a particular problem description: there are no problem-instance-specific terms in the rules comprising the customization knowledge. Thus, the customization knowledge serves to "merge" the domain model for an application problem and a generic architecture to produce an application-specific architecture; the problem-class model provides the intermediate vocabulary for expressing the customization method that does the merging.

5 Constraint Checking

The customization knowledge associated with a generic architecture enables KASE to provide a systematic methodology for designing systems. Our hypothesis is that such an approach enables a designer to create a design that is relatively free from errors if the customization knowledge is correct. However, there are two ways in which errors can be introduced in the design.

First, the customization knowledge may be incorrect. In general, it is not possible to guarantee the correctness of the customization knowledge since some of it is heuristic in nature. This seems to indicate the limitations of our approach and provides a criterion for estimating the utility of the KASE approach for particular problem classes. This is discussed in greater detail in Section 9. Alternatively, such errors may be handled by an iterative process of simulating the prototype design, identifying the sources of errors, modifying the customization knowledge, and re-synthesizing the design. However, currently KASE does not contain tools to support this process.

A second way in which errors can be introduced in the design is due to the fact that KASE is based on a design assistant metaphor (as opposed to an automated designer). Thus it is possible for a designer to ignore the customization knowledge and the suggestions offered by KASE and manually customize the architecture. This may introduce errors in the design. Most design tools contain domain-independent constraints to check for the syntactic consistency of a design (e.g., each module has at least one input and output, a named procedure is not provided by two different modules). KASE contains, in addition to these, architecture-specific constraints that check for the semantic consistency of the final design. These constraints are represented declaratively in the Constraint-Checker subsystem of KASE (Figure 1). Figure 16 shows how constraints are represented in the Constraint-Checker. Details of the motivations for this representation are given elsewhere (Nakano & Bhansali, 1993a; Nakano & Bhansali, 1993b) and here we will briefly summarize the main ideas.

```
(defconstraint
        :generality <generality-type>
        :strength <strength-type>
        :designphase <designphase-type>
        :constraint <constraint>
        :annotation <string>
        )
```

Figure 16. Representation of constraints in KASE

The *:generality, :strength,* and *:designphase* attributes are used to classify constraints from different perspectives. The *:generality* attribute refers to the scope of applicability of constraints. For example, a constraint may apply to all software systems (these are the ones typically implemented in CASE tools), it may be specific to certain projects, specific to certain application domains, and so on. In KASE some of the categories for classifying constraints according to their generality are:

- *general-architectural* constraints which apply to all software designs,
- *specific-architectural* constraints which apply to all designs based on a generic architecture,
- *general-domain* constraints which apply to all problem specifications,
- *problem-class-specific* constraints which apply to all problem instances of a problem class.

The *:strength* attribute of a constraint is meant to indicate how serious the effects of violating that constraint are. Examples of constraint categories based on their strength are:

- *enforced*: These are constraints that are automatically enforced by KASE. This is implemented using attached methods on slot attributes and the active value feature[3] in KEE. Type constraints and certain kinds of inverse relations are typical constraints that belong to this category.
- *strong:* These are constraints which, if violated, would imply a fatal flaw in the design and would result in a run-time error if not resolved.
- *weak:* These are constraints which, if violated, usually indicate some redundancy or sloppiness in the design and may or may not be harmless. (If we use an analogy with compilation then the strong and weak constraints correspond to the error and warning messages, respectively that are generated by a compiler.)

The *:designphase* attribute refers to the relevance of the constraint to a particular process or phase of the design. Currently there are only two phases that are recognized by KASE - a *modeling* phase in which problem instances are described and a *design* phase consisting of

[3] The active values feature in KEE allows one to specify that a particular action always occur whenever the value of a slot is accessed or modified.

Constraint	Generality	Strength	Phase
If a module Xrequires procedure P then there must be some module Y to which X has access and which provides procedure P.	general-architectural	strong	design
If transition T1 and T2 exist for some state and the conditions which cause T1 and T2 are identical, then the next-state for T1 and T2 should be the same.	general-domain	strong	modeling
For at least one object that needs to be tracked, there must be defined at least one operation that creates an instance of that object based on input signals	problem-class-specific	strong	modeling
All procedures that are provided by modules must be required by some other module.	general-architectural	weak	design
No module must be decomposed into more that 7 submodules.	general-architectural	weak	design
Any event that is used to trigger a trackingAgent must be posted by some other trackingAgent.	specific-architectural	strong	design
Any event that is posted by a trackingAgent must be generated by some other trackingAgent.	specific-architectural	weak	design
If X is a submodule of Y, then Y must be a supermodule of X.	general-architectural	enforced	design

Table 1. Examples of some constraints implemented in KASE

activities that create the components of an architectural design. Table 1 gives examples of some constraints, and their generality, strength, and designphase.

5.1 Constraint Language

The *:constraint* attribute specifies the actual constraint. Constraints are written in a language based on first-order logic. The syntax of a constraint is as shown in Figure 17.

In addition there are the following restrictions:

- There should be no free variables in the constraint (i.e. constraints are well-formed formulae).
- Each quantified variable must be of a type that has been defined as a unit[4] in KASE (e.g. *module, procedure*). This is to ensure that the quantification of each variable is over a finite range.

[4] The basic representation scheme in KEE is based on *frames* which are called *units*.

constraint ::= <*antecedent*> => <*consequent*>

antecedent ::= FORALL (<*var*> : <*vartype*>)$^{+}$ <*mformula*> |
<*mformula*>

consequent ::= EXISTS (<*var*> : <*vartype*>)$^{+}$ <*mformula*> |
<mformula>

mformula ::= (AND <*mformula*>$^{+}$) |
(OR <*mformula*>$^{+}$) |
(NOT <*mformula*>) |
<*aformula*>

aformula ::= (<*pred-symbol*> <*term*>*)

term ::= (<*fn-symbol*> <*term*>*)

Figure 17. Syntax of constraints

- Each function symbol must be a pre-defined function available in Common Lisp (e.g. *length*) or provided by KEE (e.g. *get.value* which returns the value of a slot of a unit).
- Each predicate symbol must be a pre-defined boolean function in Common Lisp (e.g. *member*) or provided by KEE (e.g. *unitp* which checks whether its argument is a defined unit or not).

The constraint-checker allows the use of expressions like:

(attribute-name unit-ref)

as syntactic sugar for the expression:

(get.values unit-type unit-ref attribute-name)

where *attribute-name* is the name of some slot of a unit of type *unit-type* and *unit-ref* is a variable of type *unit-type* or evaluates to an instance of type *unit-type*. (The *unit-type* is inferred from the declaration of *unit-ref* if it is a variable; otherwise *unit-type* is nil.)

In spite of the above restrictions, the constraint language is quite expressive and allows us to specify a wide variety of constraints including the ones shown in Table 1. As a result, evaluating constraints can be computationally very expensive if there are a large number of units in the knowledge-base. Therefore we need mechanisms that can increase the efficiency of constraint checking.

The technique that we have implemented in KASE is based on the observation that in most cases a designer starts out with a consistent state of the design, makes changes on certain parts of the design or problem specification, and then checks to see if he or she has violated some constraint. In such cases, instead of checking all the constraints known to the system, it is necessary only to check those constraints that could possibly have been affected due to the design

actions taken. (This is roughly analogous to the idea of incremental compilation versus full compilation).

In order to be able to identify such constraints, the Constraint-Checker computes a set of *triggers* for each constraint. The triggers attached to a constraint point to all actions that can potentially cause a violation of the constraint. When a designer initiates constraint checking on some part of the design, the constraint-checker uses the history of design actions performed by a user to determine the set of constraints that need to be checked based on the trigger attached with each constraint.

5.2 Edit Actions

Each action that a user performs during customization or while creating a problem specification can be decomposed into a set of basic actions that manipulate the units and slots represented in a KEE knowledge base. These basic actions are called *edit actions*, since they are used to edit a problem specification or design. All the allowable edit actions are modeled as part of the KASE environment. Associated with each edit action is a set of KEE/Lisp functions and predicates called *affected-clauses*. These are the sets of functions and predicates whose values might be modified as a result of executing the edit action. For example, consider the following edit-action:

edit-action: *(add-value ?utype ?u ?s ?v)*

affected-clauses: *(get.values ?utype ?u ?s)*

(has.value.p ?utype ?u ?s ?w)

(add-value ?utype ?u ?s ?v) adds *?v* to slot *?s* of a unit *?u* of type *?utype*. As a result of this action the result of the function call *(get.values ?utype ?u ?s)* will change. Similarly, the value of the predicate *(has.value.p ?utype ?u ?s ?w)* - which checks if *?w* is one of the values of slot *?s* of a unit *?u* of type *?utype* - might change - depending on what *?v* is. Consequently these two are the affected clauses of the edit action.

The edit actions are used to form triggers for the constraints.

5.3 Trigger generation

In order to explain how the constraint-checker computes triggers, we will consider a simple constraint in the following abstract form:

(FORALL (?x : T_1) (P ?x) => EXISTS (?y : T_2) (Q ?x ?y))

The compiler first translates the above constraint to the following equivalent functional form which can be interpreted by the underlying Lisp/KEE environment:

```
(if (P ?x)
    (some #"(lambda(?y) (Q ?x ?y) )
          (instances T2)))
```

where (instances T_2) denotes all instances of type T_2. In general, the translation involves dropping the universal quantifiers, substituting the existential quantifier with the some construct (available in Common Lisp), and translating the various predicates and functions to their "de-sugared" versions. Note, that as a result of dropping the universal quantifier, the constraint now contains *?x* as a free variable.

Next, the constraint checker extracts all function calls and predicates which might be affected by one of the edit actions in KASE. These are simply those function calls and predicates that unify with one of the *affected-clauses* of an edit action. Let $(P\ ?\overline{x})$ be a function call or predicate in a constraint C ($?\overline{x}$ denotes a set of variables) , $(P\ ?\overline{y})$ be an affected-clause of an edit action E, and σ be a substitution such that:

$$(P\ ?\overline{x}) = (P\ ?\overline{y})\sigma$$

Then $E\sigma$ is a trigger for the constraint C. At run-time when the constraint-checker is initiated, it tries to match the trigger, $E\sigma$, with a step in the design history. Let E' be a design step and θ be a substitution, such that

$$(E\sigma)\theta = E'$$

The constraint checker will then evaluate the constraint $C\theta$. If there are any free variables in the constraint, the constraint needs to be evaluated for each possible instantiation of each of the free variable. Also, note that any substitutions involving the bound variables of the constraint can be dropped from θ. This follows directly from the axioms of lambda calculus:

$[?z/M](\lambda\ ?z.\ (f\ ?z))\ N)$

$=$ $[?z/M](\lambda\ ?z'.\ (f\ ?z'))\ N)$ *; Renaming z to z'*

$=$ $(\lambda\ ?z'.\ (f\ ?z'))\ N$ *; Since ?z does not occur free in N*

$=$ $(\lambda\ ?z.\ (f\ ?z))\ N$ *; Renaming z' to z*

The next section illustrates the constraint checking mechanism using an example.

5.4 Example: Constraint Checking

Consider the sixth constraint in Table 1 which involves the m-TrackingAgent module of our generic Tracking architecture. This module provides three procedures: get-triggers, get-action, and get-posted-events. get-action returns a procedure (hereby called the *tracking action)* which is an instance of the *tracking-operation schema* (Section 4.1). get-triggers returns a set of events signifying when the tracking action can be executed. get-posted-events also returns a set of events; these are the possible events that can be posted by the tracking action.

A common kind of error that is made in defining m-TrackingAgent modules is that an event that is used to trigger a tracking action is not posted by any procedure. Therefore we need a constraint to safeguard against such errors. In order to represent such constraints, we introduce two new parameters for the m-TrackingAgent called triggers and posted-events. The value of triggers

is the (constant) output of get-triggers and the value of posted-events is the (constant) output of get-posted-events. In our notation this constraint is represented as follows:

```
(FORALL ((?M  module) (?E event))
     (AND (member ?M  (submodules 'm-TrackingAgent))
          (member ?E (triggers ?M)))) =>
(EXIST ((?Mprime module))
     (AND (member ?Mprime  (submodules 'm-TrackingAgent))
          (not (= ?Mprime  ?M))
          (member ?E (posted-events ?Mprime))))
```

KASE compiles the above constraint into the following equivalent expression in Lisp:

```
(if (and (member ?M  (get.values nil 'm-TrackingAgent 'submodules))
         (member ?E (get.values 'module ?M 'triggers)))
    (some #'(lambda(?Mprime)
                (and    (member ?Mprime (get.values nil 'm-TrackingAgent 'submodules))
                        (not (unit-equal ?Mprime ?M))
                        (member ?E (get.values 'module ?Mprime 'posted-events)))
          (CCS-collect-units 'module)
```

with the type declaration:

```
((module ?M) (event ?E))
```

Unit-equal and *CCS-collect-units* are built-in function in the Constraint-Checker. *Unit-equal* checks for the equality of two KEE units. *CCS-collect-units* takes a *unit-type* as an argument and returns the set of all units in KASE's knowledge-base that are of type *unit-type.*

To derive the trigger, the constraint checker extracts each atomic formula (a function call or a predicate) in the constraint that unifies with an *affected-clause* of an edit action. In the above constraint there are three such formulae:

(get.values nil 'm-TrackingAgent 'submodules)

(get.values 'module ?M 'triggers)

(get.values 'module ?Mprime 'posted-events)

The value of these functions can change if the submodule slot of m-TrackingAgent is changed (i.e. a new submodule is added or an existing one deleted), or if the value of the trigger or posted-events slot of a module is changed. The corresponding edit action for doing these is:

edit-action: *(change-slotvalue ?unit-type ?unit-name ?slot)*

affected-clauses: *(get.values ?unit-type ?unit-name ?slot)*

Unifying the affected-term with each of the three formulae above and applying the substitution to the edit-action we get the following triggers:

(i) (change-slotvalue nil 'm-TrackingAgent 'submodule)

(ii) (change-slotvalue 'module ?M 'triggers)

(iii) (change-slotvalue 'module nil 'posted-events) *(Note, that the substitution [?unit-name/?mprime] has been dropped).*

The above constraint will be checked when:

- the *submodule* slot of any unit called *m-TrackingAgent* is changed (for each possible binding of ?M and ?E);
- the *trigger* slot of any *module* A is changed (with ?M bound to A);
- the *posted-events* slot of any *module* is changed (for each possible binding of ?M and ?E)

(End of example)

One of the limitations of the above approach is that it depends on how completely the various edit actions have been modeled in terms of their effects on the function and predicate values used in the constraint. Currently we have only considered those functions and predicates that compute or evaluate some property of a unit or slot represented in KASE's knowledge base. Based on our experience so far, we have found that this is sufficient to detect all constraint violations of interest without having to do an exhaustive check.

It is, of course, possible for a user to force KASE to check all constraints irrespective of the history of edit actions taken by the user. In that case, the constraint-checker ignores the trigger and substitution information and tries each constraint for each possible substitution of the free variables in the constraint. Currently, there are about 30 constraints (including the architecture-specific and problem-class-specific constraints for the tracking domain). Checking all the constraints after the architecture has been instantiated takes a few minutes, whereas checking constraints based on the history of edit actions takes from a few seconds to a few minutes depending on the length of the edit action history.

The constraint checker has a rudimentary paraphraser which provides an English paraphrase of each violated constraint. A unique feature of the constraint generator is its ability to suggest remedial actions to remove the violated constraints. Details of the remedy generation algorithm may be found elsewhere (Nakano & Bhansali, 1993a).

6 Customization of Architecture: An Example Session

In order to give a flavor of the design process in KASE we will describe very briefly a design session involving a hypothetical designer who is using KASE to customize the generic tracking architecture for the tracking problem described earlier. In order to aid readability, the design

session is divided into the customization of the 4 main modules: m-TrackingArchitecture, m-Control, m-BlackboardPanel, and m-TrackingComponent. Knowledge of blackboard-model based architectures is helpful in understanding the process, but the objective is to elucidate the variety of knowledge-based assistance being provided to the designer.

6.1 m-TrackingArchitecture

The designer begins by using one of the graphical tools to show the module decomposition diagram for the generic architecture. KASE shows the customizable modules highlighted in the diagram. The designer decides to begin the customization process by starting from the top-level module, m-TrackingArchitecture. To customize the module the designer moves the mouse over the module and clicks. KASE presents a customization menu that is context-sensitive and contains a list of all known customization options available for this module, along with an explanation of what each command does on the bottom panel of the screen.

For m-TrackingArchitecture there is just one parameter called *solution-strategy* that represents the overall solution strategy for the problem. In general, there are three main strategies for solving problems in this architecture: event-driven (or data-driven or bottom-up), expectation-driven (or model-driven or top-down) and hybrid (i.e., both event- and expectation-driven). KASE presents a list of these three alternatives and asks the designer to select one. The designer decides to initially build a purely event-driven system. KASE incorporates this choice and marks the module as being customized.

There are two enforced constraints attached with this parameter that constrain the customization choices for two other modules: m-SituationBoard and m-TrackingComponent. Specifically, the constraint states that if an event-driven solution strategy is chosen then the m-ControlPanel submodule of m-SituationBoard should be of type m-EventPanel and the m-TrackingAgent submodules of m-TrackingComponent should be of type m-Event-based-TrackingAgent. However, it is not necessary for a designer to consider all the ramification of this decision at *this* point and she continues on.

6.2 m-Control

The designer next decides to work on the m-Control module. Thus, *KASE does not prescribe a predetermined sequence of design actions, and lets the designer control the design process as much as possible.* The m-Control module contains the top-level driver routine for the architecture called p-simple-control (Section 4). The algorithm essentially consists of a loop where in each iteration, the algorithm picks pairs of a tracking agent from m-TrackingComponent and an object from m-SituationBoard, and executes the operations associated with the tracking agent on the objects. Depending on what algorithm is used to select the tracking agents and objects, and how

many operations and objects are to be processed in each iteration, a wide variety of control algorithms are possible.

One of the parameter for the p-simple-control procedure is called the *focusing-strategy*. There are two choices for this parameter: *TrackingAgent-based* and *BlackboardLevel-based*. The designer chooses a tracking-agent based focusing strategy. This triggers a transformation that refines the program-schema forming the body of p-simple-control:

```
loop[{ par[assign[agents, determine-executable-agents[]],
        assign[nodes, determine-BB-nodes[]]],
     assign[schedule,determine-schedule[agents,nodes]
     call[execute-schedule[schedule]]},
      termination-condition[]]
```

```
loop[{ assign[agents, determine-executable-agents[]],
        assign[nodes, determine-BB-nodes[]]],
        assign[schedule,determine-schedule[agents,nodes]
        call[execute-schedule[schedule]]},
       termination-condition[]]
```

Each of the subroutines in the above procedure are also generic procedures that can be further customized. The user next clicks on determine-executable-agents. One of the parameters of this procedure is the algorithm used to select the list of executable tracking-agents (after having determined which tracking-agents have their trigger conditions satisfied). One choice for this algorithm, which is selected by the user, is a best-first selection algorithm. KASE does not possess enough customization knowledge to completely synthesize a best-first algorithm. So it simply records this decision and informs the user that she needs to provide an algorithm that takes as input a set of available tracking-agents and returns the most promising one.

At this point, the designer decides to shift her attention to the m-TrackingComponent module (realizing, opportunistically, that she first needs to determine the set of m-TrackingAgent submodules in the m-TrackingComponent module before beginning to design a best-first selection algorithm). Recent studies (Guindon, 1990) have provided empirical evidence that this kind of opportunistic shift occurs frequently during design and a guiding theme in our project has been to provide a design environment that permits a designer the flexibility to navigate among different components of the design (Guindon, 1992).

6.3 m-TrackingComponent

The m-TrackingComponent module contains as parameters the submodules of type m-Event-based-TrackingAgent. Each such submodule is comprised of three procedures - get-action, get-triggers, and get-posted-events - described earlier. The parameters for an m-Event-based-TrackingAgent are therefore the *trigger*, the *action*, and the *posted-events*. The user selects a customization command to suggest the set of tracking actions to be used in instantiating each m-Event-based-TrackingAgent. KASE responds with the following message:

"You need to first instantiate the Blackboard-Levels parameter of m-BlackboardPanel module!"

KASE uses the representation of customization commands to detect dependencies between design steps and is able to detect and warn a user if the user tries to customize a parameter whose value might later be affected by another parameter. Thus, *although KASE does not prescribe a particular design process, it warns a designer if she tries to initiate a design step that is likely to be revised later when some other part of the design is instantiated.*

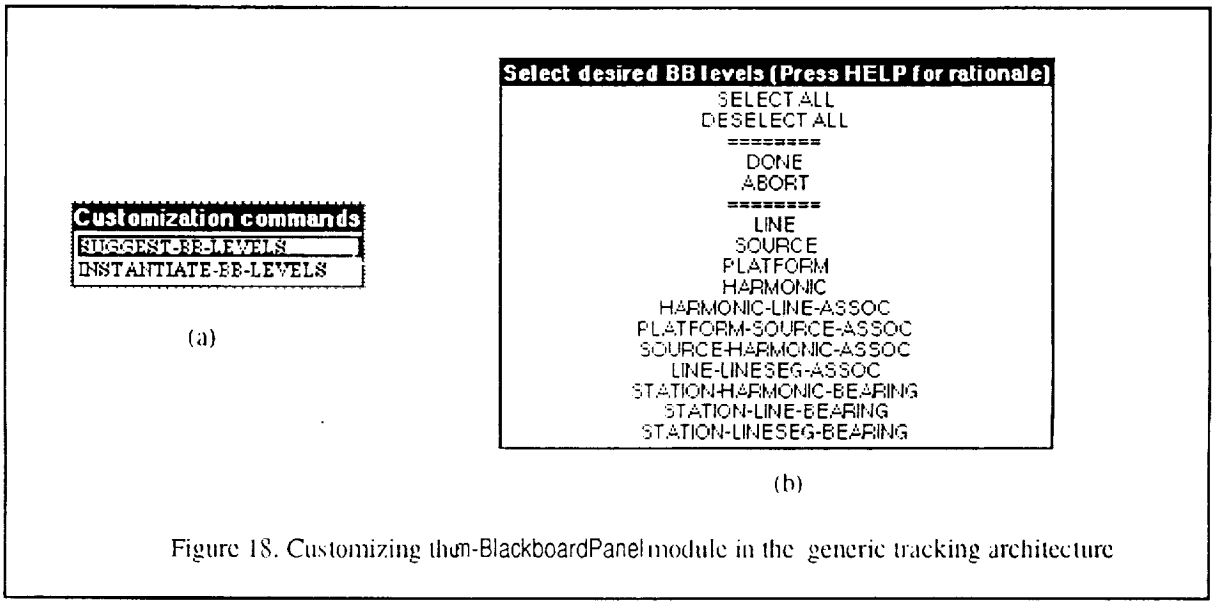


Figure 18. Customizing them-BlackboardPanel module in the generic tracking architecture

6.4 m-BlackboardPanel

Guided by KASE the designer proceeds directly to the m-BlackboardPanel module. The parameters here are the submodules m-Level each of which contains procedures to create and manipulate instances of some object or relation. The customization commands available for this module are shown in Figure 18(a). Clicking on `suggest-bb-levels`, the user is presented with a list of objects and relations that should be represented in m-Levels (Figure 17(b)). The designer can ask KASE to explain its suggestions and KASE uses annotated text templates

associated with the rules to provide explanations, for example (the following is a verbatim copy of an explanation generated by KASE):

"LINE is an intermediate object/relation needed in order to track
PLATFORM
SOURCE
SOURCE-LINE-ASSOCIATION.
Therefore, it must be a level of m-BBPanel"

The designer can use this rationale to modify his requirements and/or refine a KASE design heuristic. (This would involve changing the rule or method associated with the customization command. Currently KASE does not contain any tool to help a user in doing this.)

6.5 m-TrackingComponent Revisited

Having instantiated the submodules of m-BlackboardPanel parameter the designer returns to the m-TrackingComponent module and re-tries to instantiate its submodules. The tracking actions consist of all operations required to compute and monitor the various properties of the objects and relations to be tracked.

KASE first determines the set of all operations that can affect any of the objects or relations represented in m-Level. It then determines the set of events for each of the operations, using a set of heuristic rules (a paraphrase of some of the heuristics are shown in Figure 19). A designer can ask KASE for a rationale regarding what operations an event triggers and why, which operations post an event and why, and why a particular operation was selected to be a tracking action.

Heuristic 1 (Determining types of events).

If an object is represented in an m-level , then create events for each attribute of the object that can be modified. (The event represents the fact that the value of the object attribute has been updated).

Heuristic 2 (Determining preconditions)

If an operation, Op_1, updates the value of a derived attribute, A_1, and the value of the derived attribute functionally depends on the value of some other attribute, A_2, then any event that signals an update in the value of A_2 must trigger operation Op_1.

Figure 19. Examples of heuristics used to determine the set of events triggering trackingAgents.

There are other customization commands provided by KASE that automate some of the more frequently occurring design activities for such architectures, for example, design optimizations. One such optimizing command is to merge events. It may be the case that whenever a particular event occurs it is usually accompanied by another event. For example, the change in a particular attribute, say heading, of a tracked object may usually be accompanied by changes in its velocity

as well as a frequency shift in the signal associated with that object. Thus, it might be more efficient to group all operations that depend on either of these three events and perform them together.

This example illustrates another guiding theme of our approach that is well-known in knowledge-based software engineering research (e.g., (Smith, 1990; Waters, 1985)): *Divide the design task between a human and KASE in a way that exploits the unique skills of each.* In general, the human is better equipped to decide when to apply an optimization technique and what optimization techniques to use, whereas the machine is better equipped to carry out the optimization task, propagate the effects of those changes to other parts of the program (in the above example revising the *trigger* and *posted-events* parameter of each tracking agent), remember the optimization task, and if necessary, undo the effects of the optimization operation later. The use of generic architectures provides a context whereby useful and common *architecture-specific* optimization tasks can be identified and mechanized.

6.6 Constraint checking

At this point the designer wishes to see if the design so far violates any constraints. So she initiates the constraint checker. (For the sake of illustration we will assume that the designer has ignored one of the suggested values for the trigger parameter of m-TrackingAgents). The constraint checker provides to a user two panels called the *Constraints Filtering panel* and the *Edit Actions Filtering panel* which allow a user to limit the set of constraints checked based as well as to look for only those constraints that get violated as a result of a specific set of edit actions (Nakano & Bhansali, 1993a). The user clicks on *Strong* constraints, *Specific architectural* constraints, and *Designphase* in the *Constraints Filtering panel* and selects all the actions from the *Edit Actions Filtering* panel.

The constraint checker presents to the user one design violation. The violation is caused due to the following constraint: *An event that is posted by a trackingAgent must be generated by some other trackingAgent.* The diagnostic message generated by the constraint checker is:

Violation: V1, Constraint=Trigger-generated ,
Vars=(?M TrackingAgent-231) (?E line-segment-created-event)
Explanation: The event line-segment-created-event which is used to trigger trackingAgent-231 is not posted by any other trackingAgent.

The designer may now examine the tracking action associated with *TrackingAgent-231* to see why *line-segment-created-event* is needed as a trigger. She may then decide to either delete *line-segment-created-event*as a trigger or may introduce a new trackingAgent that posts this event.

For certain kinds of constraints, the constraint checker also generates a list of suggested remedies and also computes the effect of executing each of those remedies (Nakano & Bhansali, 1993a)

7 Discussion of Results

We implemented the customization knowledge and constraints for customizing the generic tracking architecture and have successfully used them to synthesize two different systems from the same generic architecture for tracking. The two systems were (1) ELINT- designed for tracking aircraft based on radar signals emitted by them(Bhansali & Nii, 1992a), and (2) HASP - designed for tracking ships and submarines based on noise signals generated by them. Both these systems had been originally designed by different groups of designers(Brown et al., 1986; Nii et al., 1982) and both were later fielded. Our exercise using KASE essentially showed a rational reconstruction of the designs of these two systems using the same design process on a generic architectural design. When compared with the results of the original design (Table 2)[5] we found that the results produced by KASE were more systematic and (we believe) more reliable although they may not be as efficient.

	ELINT	KASE-ELINT	HASP	KASE-HASP
Blackboard Levels	3	5	4	11
Tracking Agents	6	10	13	26
Events	9	15	14	46

Table 2. Comparison of designs produced by KASE with the original designs

For example, using KASE we obtained about twice as many m-TrackingAgent modules as in the original design of HASP. The tracking agents differed considerably in their triggers. Comparing these tracking agents with the original design, we found that several trigger conditions were either missing or were unnecessary in the original design. But they had been designed in this way in order to reduce the overhead of scheduling knowledge sources: combining the functionality of several tracking agents into one module and triggering them based on a small set of triggers results in faster scheduling of the tracking agents in each control cycle.

[5] We actually implemented only a subset of the complete domain of ELINT and HASP. The numbers for ELINT and HASP in Figure 12 represent the numbers when restricted to those subsets (which consists of about eighty percent of the complete system).

Using the optimization commands mentioned earlier (Section 6.5) it is possible to obtain the same design as in the original implementation; however, the important point is that by using KASE one can systematically consider the implications of each heuristic that designers normally use in designing systems. This reduces errors in the design due to omission. Furthermore, by recording the history of customization steps (which is done automatically by KASE) a designer can re-create or undo the design process at a later stage. This enhances the maintainability of the system.

We also conjecture that besides improving reliability and maintainability, KASE improves productivity. We are not yet in a position to support this conjecture directly since we do not have data which can be used to compare the effort expended by the original designers in designing their systems and the effort expended in designing them using KASE. However, we can obtain indirect evidence by comparing the effort expended in designing ELINT and HASP using KASE. The design of KASE-ELINT took about 5 months (including the acquisition of domain and customization knowledge and implementation of the optimizing commands) whereas the design of KASE-HASP took only 2 months (of which a major part was devoted to domain knowledge acquisition which was considerably larger that ELINT's and not well documented). In other words, after the first design using a generic architecture, the effort for subsequent designs is determined mostly by the effort in domain knowledge acquisition.

KASE has been implemented using Lisp and KEE, an object-oriented knowledge representation and programming environment. It contains about 15000 lines of Lisp code and runs on TI Explorers. Except for some of the graphical display routines the rest of the code is quite efficient. The output that is produced by KASE is a mix of specifications (using pre- and post-conditions) and pseudo-code. For the two systems that have been produced so far, it is relatively straightforward to convert this output to code. In our current work (see below) we have added transformation rules that converts the output of KASE to produce executable code (in Fortran).

8. Current and Future Work

KASE is not yet an industrial-strength system. It has so far been used only by members of the KASE project. However, different parts of the system were implemented by different people and are constantly in use by other members of the project. This offers us confidence in the robustness and usability of the system.

Our current work seeks to measure the generality and usability of KASE by using it in two different domains. The first domain is concerned with a subsystem of KASE itself: the diagram manager subsystem. This system contains the routines that implement the graphical interface of KASE. It is used to display the problem-class model and the architecture structure through

various diagrams (e.g., module decomposition, data-flow, and state-transition diagrams). These routines were written by different members of the KASE project at different times and differ considerably in their implementation details. However, they can all be described in a uniform manner at some architectural level. We have created a generic architecture that represents the design of the various subsystems and have implemented the customization knowledge that can be used to synthesize the individual (existing, as well as new) diagramming subsystems semi-automatically. Using the system we have been able to generate new kinds of diagrams in minutes instead of a few days that it used to take earlier (Bhansali, 1993).

The second domain we are investigating is concerned with the analysis of radio signals obtained by planetary probes (e.g., Voyager and Mars Observer). We currently have an architectural design of a system that processes these signals and performs various kinds of filtering, compression, and error-correction on them and computes various geometrical properties of celestial bodies. Depending upon various parameters like the path taken by a signal, the accuracy required, or the throughput needed, different variants of the system are created by individual users. Our goal is to represent the generic design of all these systems in KASE as a generic architecture and investigate how KASE can help the users in improving their productivity. Although we do not believe that KASE is ready to be made available for general use soon, we are planning to have actual end-users use the system and obtain empirical evidence to validate its usability.

9. Related Work

We share the general goal of supporting the synthesis of domain-specific software systems with many other projects (e.g. KITSS(Nonnenmann & Eddy, 1992), SINAPSE(Kant,Daube,MacGregor, & Wald, 1991), ELF(Setliff & Rutenbar, 1992), ΦNIX(Barstow, 1983)) although the various projects differ substantially in their respective approaches. Some of these differences are due to the different domains being addressed (e.g. telephony, CAD, scientific computing) , the generality of the class of systems addressed, and the exact form of the operational goal (code generation, specification acquisition, design, testing, etc.) of the projects. A representative sample of some of the other domain specific software systems can be found elsewhere(Lowry & McCartney, 1991). Here we briefly survey some of the more closely related work and how they relate to our project.

KASE is based on a novel framework for developing software systems in which generic architectures are the fundamental unit of reuse. In this respect our work is related to the Domain-Specific Software Architecture (Workshop, 1990) project. However, as far as we know, KASE is the first system that has (1) demonstrated how the concepts of software architectures, problem classes, and software synthesis can be integrated in a unified framework, and (2) shown the

applicability of the framework in the design of two different systems based on a common generic architecture.

If we consider the generality of domain-specific software systems, then on one end of the spectrum lies the application generator approach. Application generators may be thought of as high-level compilers for narrow-spectrum, application-specific languages. They are well suited for domains where a set of requirements can be easily expressed in some simple, high-level language. However, since the knowledge about the application domain is embedded in the macros and interpreters of the application generator and the compilation process is opaque to an end-user, it is difficult to adapt them for different application. In KASE, the knowledge for customizing an architecture is represented explicitly as rules and methods which makes it much more flexible than application generators. On the other hand, it seems to indicate that the KASE approach is not appropriate for all problem classes. Its utility depends critically on the complexity of acquiring and representing the relevant customization knowledge and ensuring its correctness. At one extreme, there are classes of problems that are well understood and for which the customization knowledge would be relatively complete and correct. In such domains an application generator approach would be more efficient than KASE. On the other extreme are entirely new classes of problems for which little is known regarding the design process. In such domains, the customization knowledge would be very sparse and KASE could not offer much assistance beyond that offered by the current CASE technology. Thus, it seems that KASE would be most useful for domain that lie between these two extremes.

The idea of constructing software systems by first capturing a model of a class of systems was first presented in a system called Draco (Neighbors, 1984). In the Draco approach, there exists a hierarchy of domains each corresponding to either a specific problem area (called an *application domain*) or a general, application-independent domain (called a *modeling domain*). The application domain of Draco corresponds to a problem-class description in KASE. The major differences between the Draco approach and KASE arise due to the different interpretation on what constitutes the basic unit of reuse. In Draco, the various domain models are the basic reusable units. The design task consists of refining problem specifications written in one domain language into another domain language repeatedly until an "executable" domain is reached. On the other hand, in KASE our emphasis has been less on the reuse of domain models and more on the reuse of software architectures, which is considered the basic reusable unit. The design task consists of refining a generic software architecture into a specific one suitable for a specific problem instance, and the major concern is on providing tools and mechanisms that allow software designers to do it efficiently.

Software architectures are the focus of considerable attention by several researchers at Carnegie Mellon University (Allen & Garlan, 1992; Lane, 1990; Shaw, 1989). A major objective

of the work being done there is the development of an (application-independent) taxonomy of software architectures. This includes the identification of commonly used architectural paradigms, the relationship between various architectural paradigms and problem classes, and analysis of trade-offs involved in choosing one paradigm over another (Shaw, 1989). Lane (Lane, 1990) describes a user-interface software architecture and design rules for building specific user interface systems. The approach consists of creating a space of design alternatives and formulating rules that indicate good and bad design choices based on problem requirements - which is quite similar to the approach in KASE. However, it seems that there is less support for semi-automated or automated conversion of the resultant design into code. Another line of research is concerned with investigating how software architectures can be formally represented, allowing one to explore its properties systematically which in turn could ultimately lead to algorithimic techniques for choosing software architectures (Allen & Garlan, 1992).

Other closely related work to KASE are the LEAP project at Lockheed (Graves, 1991) and the ROSE-2 system developed at MCC. LEAP also uses architectures as a basis for synthesizing systems and relies on an interactive designer to synthesize a specific system. It is also capable of learning relevant rules dynamically during design. ROSE-2 contains a library of generic design schemas for certain application classes that can be composed and refined to produce specific designs for problem instances.

Most of the above systems, including KASE, are concerned with routine or parametric design whereby an abstract, general artifact is refined into a specific one. In contrast, Feather et al. (Feather,Fickas, & Helm, 1991) describe how non-routine or innovative designs can be produced for certain kinds of applications that involve multiple agents interacting in order to achieve some overall functional goal. Their approach consists of creating a set of design operators that is sufficient to create a search space of all possible designs for an application class, followed by a heuristic search in the design space for solving a particular problem. However, other than this, little work that has been done in exploring how innovative or novel software system designs can be synthesized. A good topic for future research would be to investigate principled ways of synthesizing generic software architectures from a set of basic building blocks.

There is close parallel between our approach of synthesizing software systems and parameterized programming (Goguen, 1989). In parameterized programming, a generic program represents a parameterized algebraic theory which can be instantiated by using a view (theory morphism) that binds actual and formal parameters. From this perspective, KASE can be thought of as providing a library of highly parameterized programs (the generic system design) as well as tools to assist a user in instantiating them based on the requirements of a particular application.

10. Conclusions

We have presented an approach to software reuse that is based on abstracting the design of a class of problems as a generic architecture. Such an approach provides reuse at the level of entire systems in addition to reuse at the level of algorithms or subroutines. We have proposed that a generic architecture can be usefully viewed as a cross-product of a problem-class and a set of solution features. A library of generic architectures can thus be obtained and indexed by considering various combinations of problem classes and solution features. We have proposed that knowledge about an application be separated into a problem-class which contains concepts generic to a class of problems and a problem-instance which contains concepts specific to a particular application. The problem-class model can then be used to facilitate the acquisition of the specification of a problem instance - a task that currently constitutes a significant bottleneck in creating domain-specific systems. In addition, the problem-class model can be used to formalize the design process knowledge which, in turn, can be used to assist designers in designing systems for many different application problems.

A distinguishing feature of KASE is its emphasis on providing tools that support the way humans design. The goal in KASE is not towards full automation of the design process as in traditional automatic programming research (e.g. (Smith, 1990)). Instead, KASE's goal is to supply the appropriate knowledge to a designer in the right context so as to enable him to increase his productivity and reduce errors in the design. This has led us to adopt an approach that strikes a balance between entirely formal, axiomatic approaches and informal, hypertext-based approaches. We have also striven to incorporate existing software modeling and design practices in KASE. In particular, we have described a scenario where object-oriented modeling, structural and functional decomposition, data-flow and state-transition diagrams can all be useful in the design of complex systems. This makes KASE a practical tool that can be used in an incremental manner by software designers.

The KASE approach is not suitable for all problem classes. It's utility depends to a large extent on the feasibility of acquiring the relevant customization knowledge. We believe that KASE is most appropriate for problem classes whose design process is neither well understood nor poorly understood. One of the research issues that we are interested in is can some of the customization knowledge be acquired and subsequently modified directly by end-users? One way of doing this is to infer or *learn* appropriate customization rules by observing a user's actions. Such a capability has been proposed in (Garg & Bhansali, 1992) and we plan to investigate how that work can be extended to learn design rules in KASE.

Acknowledgments

This research was supported in part by National Aeronautics and Space Administration under Grant NCC 2-749. The KASE system is the result of contributions from several people: H. Penny Nii, Nelleke Aiello, Go Nakano, Raymonde Guindon, Liam Peyton, and Eduardo Parra. The author would like to thank Michael Lowry and Tom Pressburger for reading an early draft of this paper and for their many helpful comments.

References

Aho, A. V., & Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley Publishing Company.

Allen, R., & Garlan, D. (1992). A Formal Approach to Software Architectures. In *IFIP World Computer Congress 92*. Madrid, Spain.

Barstow, D. (1983). A Perspective on Automatic Programming. In *8th International Joint Conference on Artificial Intelligence*, (pp. 1170-1179).

Bhansali, S. (1992). Generic software architecture based redesign. In *AAAI Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, (pp. 53-58). Stanford, CA:

Bhansali, S. (1993). Architecture-driven Reuse of Code in KASE. In *Fifth International Conference on Software Engineering and Knowledge Engineering*. San Francisco Bay: Knowledge Systems Institute.

Bhansali, S., & Nii, H. P. (1992a). KASE: An integrated environment for software design. In *2nd International Conference on Artificial Intelligence in Design*, (pp. 371-389). Pittsburgh, PA:

Bhansali, S., & Nii, H. P. (1992b). Software Design by Reusing Architectures. In *7th Knowledge-Based Software Engineering Conference*, (pp. 100-109). McLean, Virginia: IEEE Computer Society Press.

Brown, H. D.,Schoen, E., & Delagi, B. A. (1986). An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures No. STAN-CS-86-1136. Department of Computer Science, Stanford University.

Conklin, J., & Begeman, M. (1989). gIBIS: A Tool for all Reasons. *Journal of the American Society for Information Science*, 40, 200-213.

Feather, M.,Fickas, S., & Helm, B. R. (1991). Composit System Design: the Good News and the Bad News. In *6th Annual Knowledge-based Software Engineering Conference*, (pp. 13-27).

Garg, P., & Bhansali, S. (1992). Process Programming by Hindsight. In *14th International Conference on Software Engineering*, (pp. 280-293). Melbourne, Australia: IEEE Computer Society.

Goguen, J. A. (1989). Principles of Parameterized Programming. In T. J. Biggerstaff & A. Perlis (Eds.), *Software Reusability* Addison Wesley.

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

Gruber, T. R. (1991). The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases. In J. A. Allen,R. Fikes, & E. Sandewall (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the 2nd International Conference* San Mateo, CA: Morgan Kaufmann.

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. In *ACM Proceedings of CHI'92*, . Monterrey, CA:

Jagannathan, V.,Dodhiawala, R., & Baum, L. S. (Ed.). (1989). *Blackboard Architectures and Applications*. Boston: Academic Press.

Kant, E.,Daube, F.,MacGregor, W., & Wald, J. (1991). Scientific Programming by Automated Synthesis. In M. R. Lowry & R. D. McCartney (Eds.), *Automating Software Design* (pp. 169-206). AAAI Press/The MIT Press.

Lane, T. G. (1990). A Design Space and Design Rules for User Interface Software Architectures No. CMU-CS-90-176. Carnegie Mellon University.

Lowry, M. R., & McCartney, R. D. (Ed.). (1991). *Automating Software Design*. AAAI Press/The MIT Press.

Nakano, G., & Bhansali, S. (1993a). Flexible control mechanism in a consistency maintenance system. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*., . Victoria, British Columbia, Canada.:

Nakano, G., & Bhansali, S. (1993b). A knowledge-based approach for consistency checking mechanism in software design. In *Proceedings of the 6th Florida AI Research Symposium*, (pp. 157-165). Fr. Lauderdale, FL:

Neighbors, J. (1984). The DRACO approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(9), 564-573.

Nii, H. P., & Aiello, N. (1979). AGE (Attempt to Generalize): A knowledge-based program for building knowledge-based programs. In *6th International Joint Conference on Artificial Intelligence*, (pp. 645-655).

Nii, H. P.,Feigenbaum, E. A.,Anton, J. J., & Rockmore, A. J. (1982). Signal-to-Symbol Transformation: HASP/SIAP Case Study. *AI Magazine*, Spring, 23-36.

Nii, P. (1989). Blackboard Systems. In A. Barr,P. Cohen, & E. Feigenbaum (Eds.), *Handbook of Artificial Intelligence* (pp. 1-82). New York, NY: Addison-Wesley.

Nonnenmann, U., & Eddy, J. K. (1992). KITSS - A Functional Software Testing System using a Hybrid Domain Model. In *Proceedings of the 8th Conference on Artificial Intelligence for Applications*, . Monterrey, CA:

Prieto-Diaz, R., & Neighbors, J. M. (1986). Module Interconnection Languages. *Journal of Systems and Software*, 6(4), 307-334.

Rumbaugh, J..Blaha, M.,Premerlani, W.,Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design*. Englewood Cliffs, New Jersey: Prentice Hall.

Setliff, D., & Rutenbar, R. (1992). Knowledge Representation and Reasoning in a Software Synthesis Architecture. *IEEE Transactions on Software Engineering*, 18(6).

Shaw, M. (1989). Large scale systems require Higher-level abstractions. In *Fifth International Workshop on Software Specifications and Design*, (pp. 143-146). IEEE Computer Society.

Shaw, M. (1991). Heterogeneous design idioms for software architecture. In *6th International Workshop on Software Specification and Design*, (pp. 158-165). Como, Italy:

Smith, D. R. (1990). KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1024-1043.

van Melle, W. (1980) A domain independent system that aids in constructing consultation programs. PhD, Computer Science Department, Stanford University.

Waters, R. C. (1985). The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11), 1296-1320.

Workshop, D. (1990). Proceedings of the Workshop on Domain-Specific Software Architectures. Software Engineering Institute.