Knowledge Systems Laboratory
Report No. KSL 92-38

April 1992
Revised: July 1992

Software Design by Reusing Architectures

by

Sanjay Bhansali
H. Penny Nii

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

This work was supported in part by NASA under grant NCC 2-749-1

Software Design by Reusing Architectures

Sanjay Bhansali
H. Penny Nii
Knowledge Systems Laboratory, Stanford University
701 Welch Road, Building C
Palo Alto, CA 94304

Abstract

Abstraction fosters reuse by providing a class of artifacts that can be instantiated or customized to produce a set of artifacts meeting different specific requirements. We propose that significant leverage can be obtained by abstracting software system designs and the design process. The result of such an abstraction is a generic architecture and a set of knowledge-based, customization tools that can be used to instantiate the generic architecture. In this paper we describe an approach for designing software systems based on the above idea. We illustrate the approach through an implemented example, and discuss the advantages and limitations of the approach.

1 Introduction

Constructing 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. Abstraction fosters reuse 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, encapsulation or information hiding, and parameterized modules are examples of some of the most notable application of the abstraction principle in software systems.

The goal of artificial intelligence applied to software engineering is to provide increasing automation to the software development process. The application of the abstraction principle to automate the construction of artifacts (that would normally require a creative process) can be found in early AI work. For example, Emycin (1), an expert system shell was developed by abstracting the control structure of Mycin; abstracting out the process of building blackboard systems yielded AGE (2). Subsequent commercial expert systems shells are based on different mixtures of design and process abstractions. More recently, abstraction has been successfully used in automatic programming: the KIDS system (3) 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 (4, 5, 6) we are currently investigating the utility of abstracting *software system designs and the design process*. It is generally conceded that designing software systems is a creative and ill-understood process. Successful software designs are created by a very small group of designers; however, the process is rarely documented and the final design is typically not well documented. Consequently it is very 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, (2) abstracting the design of the system as a generic architecture, and (3) building tools that allow specific systems to be constructed semi-automatically by customizing the generic architecture. Such an approach allows us to (1) reuse the architecture for multiple applications within the class, (2) capture the process of software design which could be used to maintain the system (7) or be reused for multiple designs, and (3) ultimately learn algorithmic descriptions of the design process (8).

As an initial test-bed we chose an experimental system, ELINT (9), that was designed several years ago at the Knowledge Systems laboratory. In an earlier paper (5)we reported our initial work on KASE which provides knowledge-based support to partially automate the construction of the ELINT system using a generic architecture. In this paper we describe the reuse of the generic architecture *and* the design process to design another system, HASP (10), which is similar to ELINT, but was developed for a different domain by another team of designers. We discuss the advantages and limitations of this approach, and discuss issues for further research.

2 Generic Architectures

We first explain the notions of a generic architecture and the process of customizing the architecture more precisely.

Definition. A *module* is a packaging of objects, relations, types, and procedures in a logical unit.

In KASE, a module itself is represented as an object with a set of attributes. Figure 1 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 data-types that form arguments to procedures provided by the module and the results of procedures required by the module). A module interface is defined in terms of the services (operations, procedures) that it provides to other modules, and the services it requires from other modules. The other attributes constrain the way a system is structured and the way modules communicate with each other. For example, a module may only use services provided by its submodules or a module that it has access to.

MODULE	
supermodule	*module that contains this module*
submodules	*modules contained within this module*
provides	*services provided by this module*
requires	*services required by this module*
has-locally	*local data types and procedures*
has-access-to	*modules which can provide services to this module*
*inputs	*data flow into the module*
*outputs	*data flow out of the module*
*calls	*modules called by procedures within this module*
*called-by	*modules that call procedures provided by this module*

Figure 1. Minimal internal representation of a module.

Definition. A *parameterized module* is a module in which some of the attributes are abstracted as parameters.

A parameterized module is represented by two additional attributes: *parameter-list* and *customization*. The *parameter-list* contains a list of attributes that need to be instantiated. We allow any of the primitive attributes (except *supermodule*) to be a parameter. *Customization* contains a set of *customization commands* that can be used to customize (instantiate) a parameterized module.

Definition. A *customization command* is a tuple <P, M, F, R, [E]> where

P is the name of a parameter,

M is a method for instantiating the value of the parameter,

F is the input domain for M, i.e., the set of factors that affect the outcome of the method.

R is the output domain of M, and

(optional) E is an explanation or rationale for the customization method.

Depending upon the parameter the customization method, M, may consist of selecting a value from a precomputed list of alternatives, transforming an abstract program schema using a set of transformational rules, or inferring a value using a set of heuristic design rules and/or algorithms. Likewise the input and output domain could range from a library of reusable instances to a set of rules to a set of domain-specific assumptions. The optional argument E is a canned text string or a text template (which is instantiated based on the context) that can provide an explanation for the values computed using the customization method.

Definition. A *generic architecture* is defined as a topological organization of a set of parameterized modules, together with the inter-modular relationships.

Designing a software system using a generic architecture consists of instantiating the parameters of each parameterized module by a concrete value while maintaining the inter-modular constraints.

Figure 2 shows an overview of the design process based on generic architectures in KASE. The boxes outlined with double lines represent knowledge components that are part of KASE. 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 3.1). Associated with the generic architecture is a *meta-model* which may be thought of as a representation scheme for a problem class (section 3.2). A particular application problem is described by instantiating this meta-model (section 4). The final knowledge component called *customization knowledge* contains the knowledge necessary to customize the generic architecture and is the basis of KASE's intelligent support (section 5).

3 Architectural commitments

In any generic architecture, the modules and inter-modular relationships exist within a semantic context consisting of (1) goals and subgoals of a task and a strategy for achieving the goals, and (1) classes of objects manipulated within the architecture. Thus, when a designer selects and commits to a generic architecture for customization, he is also making two additional commitments: *task commitment* and *ontological commitment*.

3.1 Task commitment

The first commitment inherent in an architecture is the classes of problems being addressed and the overall solution strategy. Architectures are designed to solve a particular class of problems. For example, systems that perform batch transformation on a single set of inputs would have a different architecture from one that performs continuous transformations; these two architectures would be radically different from, say, a real-time interactive system that is governed by strict timing constraints and user interactions.

Architectures also embody a set of high-level, strategic decisions on how to decompose and solve the problem. Examples of such strategic decisions are whether to use concurrency or not, whether to use symbolic processing or a statistical analysis of data, whether the system functionality should be decomposed into a set of horizontal layers or whether to use weakly coupled vertical partitions. In general , these solution strategies are governed by the problem classes.

For the generic architecture currently represented in KASE, the task is to interpret continuously received signals from one or more sources and infer the activities of the moving objects. The solution strategy is to solve the problem by symbolic interpretation of the data using a blackboard problem-solving model (Nii, 1982) on a uniprocessor machine.

3.2 Ontological commitment

An ontology is a vocabulary of representational terms for describing a domain. Having associated a generic architecture with a problem class, we can create an ontology of generic terms that are relevant for describing problems belonging to that class. The ontology of generic terms are based on the conceptual primitives available in a model representation language. The modeling primitives that we have is based on an object-oriented scheme that creates a model from three different perspectives – a static or object model, a behavior model, and a functional model (11). The modeling primitives currently available in KASE are *objects, relations, operations, states, events, transitions, and (data) types.*

We call the ontology of generic terms associated with a generic architecture a ***meta-model***, which refers to the classes of objects and tasks that are intrinsic to the architecture. Figure 3 shows fragments of the meta-model associated with a generic architecture for tracking mobile platforms based on a symbolic interpretation of signals emitted by the platforms.

We assume that the ontological commitments in a particular generic architecture are shared by domain modelers and designers reusing the architecture. The use of textual annotations, mnemonic names, and explicitly represented constraints are used to facilitate the sharing to some extent. The issue of how to communicate ontological commitments in general is an important

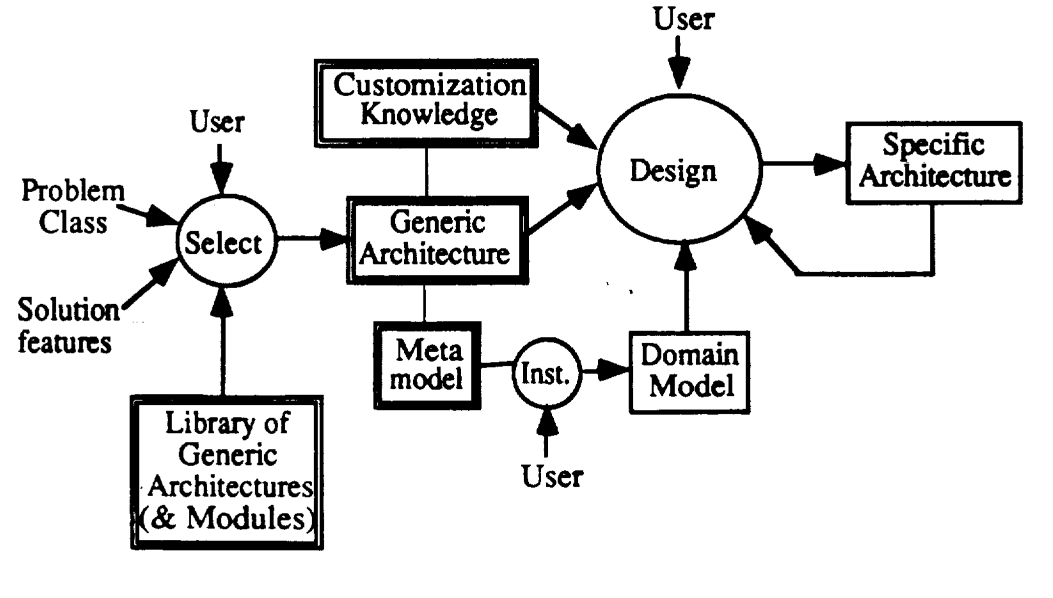
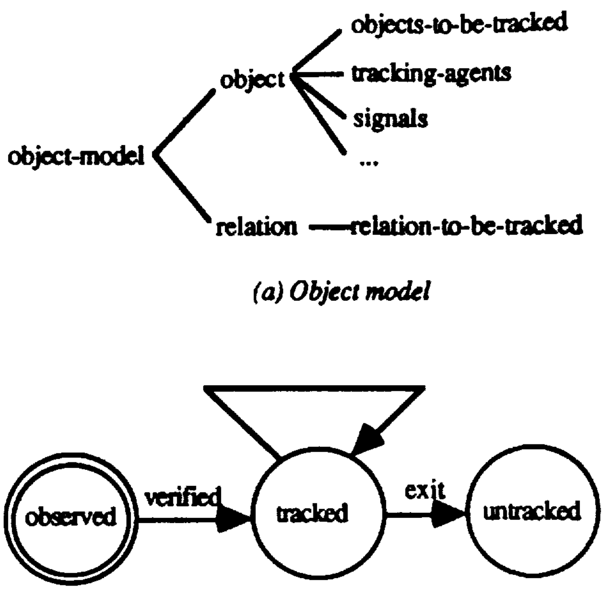


Figure 2. Overview of software design based on generic architectures in KASE.



(a) Object model

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

Figure 3. Fragments of the object(static) and behavior meta-model associated with a generic architecture for tracking objects.

research topic that is beyond the scope of our work (12).

4. Domain Modeling

The meta-model associated with a generic architecture can be used to drive the acquisition of a specific domain model. KASE provides visualization tools and editors (built on top of the KEE environment) to aid a domain analyst in instantiating the meta-model with terms and concepts of a particular domain. Figure 4 shows the instantiation of two meta-model objects in two different domains – ELINT (tracking aircrafts based on processed radar signals) (9) and HASP (tracking ships based on processed sonar data) (10).

A domain model contains the objects and relations, their attributes, and operations that need to be defined by the analyst. The attributes of a typical domain object and the operations associated with the object are shown in Figure 5. An operation is specified formally using pre- and post-conditions on the inputs and outputs o: the operation. The generic state transition diagrams associated with the objects-to-be-tracked object is used to provide a starting point to a domain analyst in specifying the operations. Typically, the domain analyst would copy the generic state transition diagram associated with an object and then modify it using the graphical and text editors provided by KASE. Once the state transition diagram is customized, the designer would fill in the definitions of each operation. Figure 6 shows the customization of a state-transition diagram for the source object and the definition of an operation on it.

One of the implications of providing intelligent assistance is that the system be able to deal with incomplete and inconsistent information (13, 14). In KASE the user is not prescribed to follow a particular sequence of steps nor is it necessary to provide complete and consistent requirements at all times. For example, the following possibilities can occur in KASE:

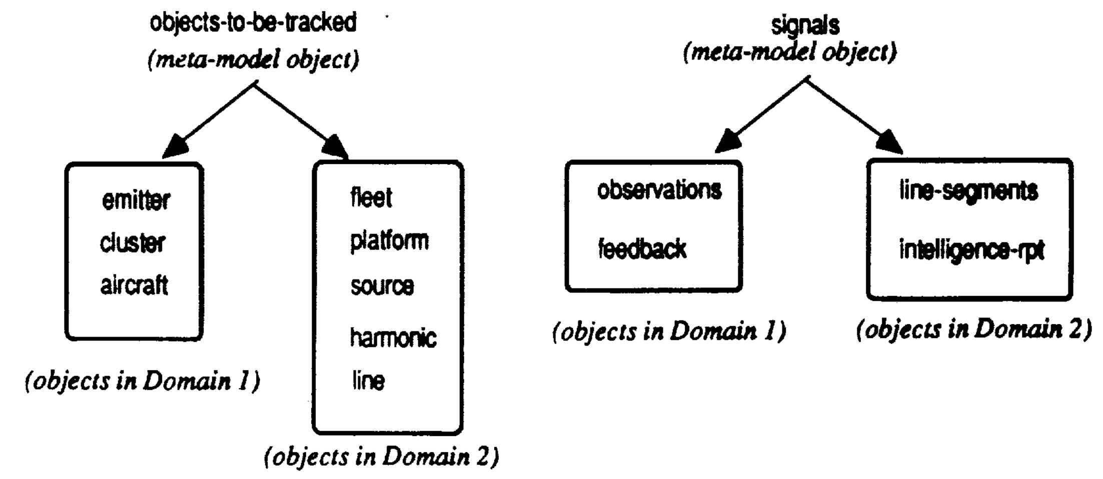


Figure 4. Instantiation of meta-model objects to object instances in two different domains.

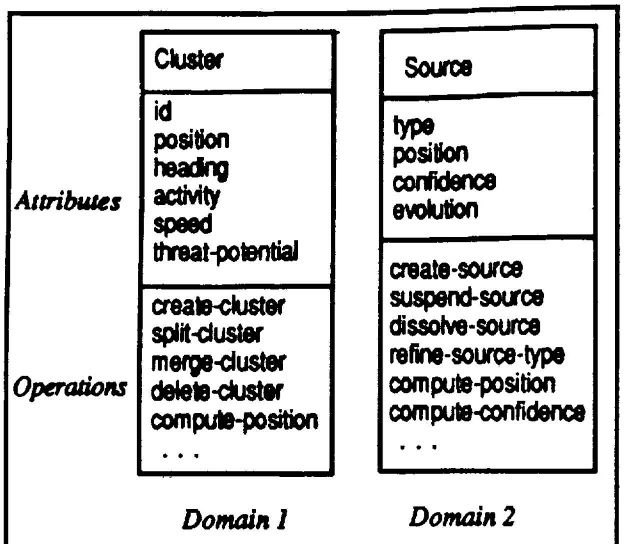


Figure 5. Attributes and operations of two typical objects in two different tracking domains.

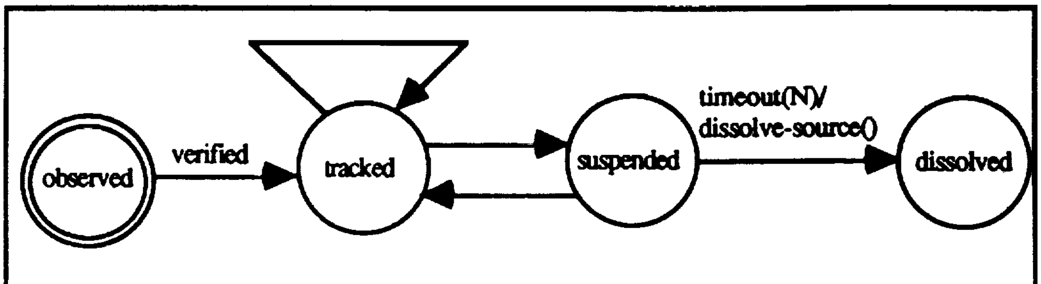
- A state transition diagram is inconsistent with the operation definition: the preconditions mentioned in the operation definition does not include an event specified in the state transition diagram, or includes an event not specified in the state transition diagram. KASE detects such inconsistencies and reports them to the user.

- The state transition diagram and the operation contain redundant information. KASE checks to see that the pre- and postconditions in the state transition diagram and the operation definition are consistent.

We created the domain model for ELINT and HASP by instantiating and extending the generic tracking meta-model. The HASP domain model consists of 6 objects to be tracked, 7 relationship between these and other objects in the domain, and 29 operation definitions, and we found that even for this small domain the constraint checking provided by KASE was very useful.

5. Customization of Architecture

Once an initial model of an application domain is in place, a designer can begin designing the system by instantiating the module parameters comprising the generic architecture. In order to illustrate the customization process we give below a description of a session involving a hypothetical designer who is using KASE to customize the generic architecture for the HASP domain. Instead of giving a detailed description of the



(a) Instantiation of a generic state transition diagram (see figure 2)

```
(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 (= ?s.state SUSPENDED)
                    (> (- (suspension-time ?s.evcode) ?t)
                       N)
                    (= ?s.evcode EVCODE)))
 :body ()
 :postcondition (AND (= ?s.state dissolved)
                     (= ?s.evcode (EVCODE || [dissolved-at ?t])))
```

(b) Definition of the dissolve-source operation appearing in the above diagram

Figure 6. Specifying a domain model by instantiating a meta-model

entire customization process, we will concentrate on highlighting some of the interesting features of KASE including the following:

- Context-sensitive customization
- Integration of different customization methods
- Suggest-and-instantiate design paradigm
- Propagation of design decisions
- Opportunistic design support
- Rationale/explanation of design process

In order to aid readability, the design session is divided into the customization of the 4 main modules: m-Signal-Interpreter, m-Control, m-BBPanel (a subcomponent of m-Situation-Board), and m-Tracking-Component (see below). Knowledge of blackboard architectures is helpful in understanding the process, but the objective is to elucidate the variety of knowledge-based assistance being provided to the designer.

5.1 m-Signal-Interpreter

The designer begins by using one of the visualization commands to show the module decomposition and data flow diagram for the generic architecture (Figure 7). The designer decides to begin the customization process by starting from the top-level module, m-Signal-interpreter. 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 this module there is just one parameter 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. At the same time it propagates the effect of this decision to the other two affected modules, m-Situation-Board and m-Tracking-Component. The overall solution strategy results in the instantiation of a few procedures in these two modules (for recording and manipulating events) and are used to customize the values of some other parameters. However, the designer need not be concerned with all the ramification of this decision at this point and continues on.

5.2 m-Control

The designer next decides to work on the m-Control

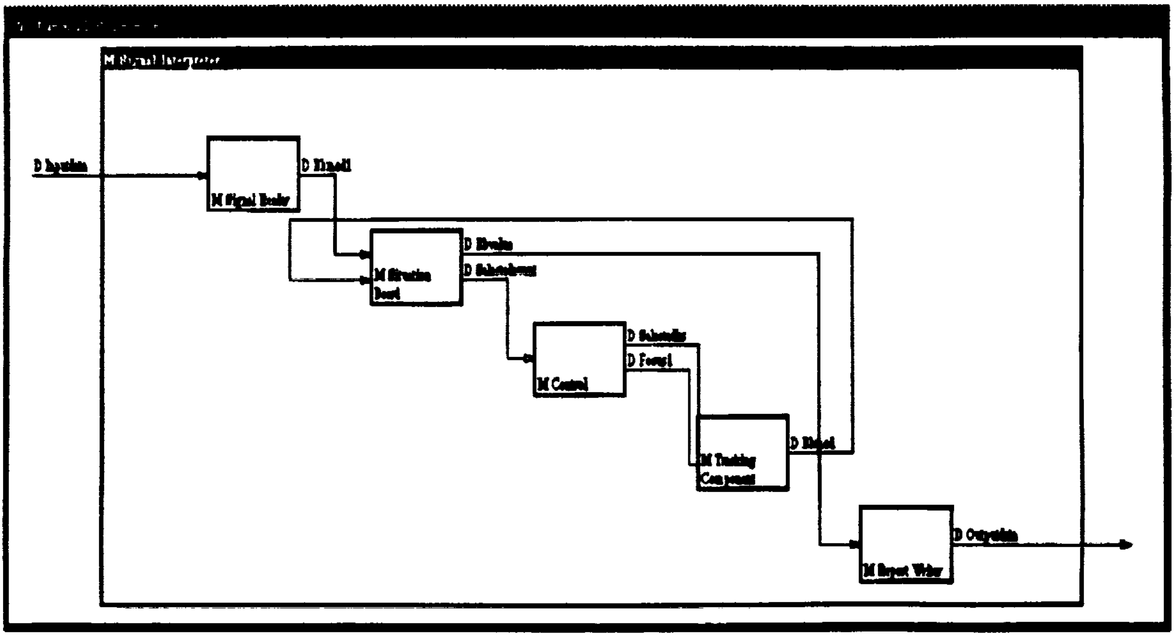


Figure 7. Module decomposition and data flow diagram of the generic architecture for signal interpretation. For simplicity only 1 level of decomposition is shown although KASE allows upto 3 levels to be shown.

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. The algorithm essentially consists of a loop where in each iteration, the algorithm picks a pair of tracking agent from m-tracking-component and some object (or a set of objects) from m-Situation-Board, and executes the operations associated with the tracking agent on the objects. Depending on what algorithm is used to select the tracking agent and object(s), and how many operations and objects are to be processed in each iteration, a wide variety of control algorithms are possible. Most of the functionality of m-Control is divided into two submodules: m-Selectfocus and m-Scheduler. m-Control itself has a code template associated with it which (when completely instantiated) contains calls to the procedure provided by the two submodules.

For the m-control module KASE displays two customization commands relating to the following parameters:

- Processing-priority: How to resolve conflicts in a hybrid processing strategy (Since the designer chose a pure event-driven processing strategy, this parameter is irrelevant).
- Focusing-strategy: How to determine the next focus of attention, i.e., should it based on the tracking agents or on the nodes in the situation board.

The designer chooses a tracking-agent based focusing strategy. This triggers a set of transformations that refine the code template associated with the m-control module and instantiates a few additional procedures provided by the submodules of m-control. Continuing with his top-down design paradigm the designer now begins customizing the submodules of m-control:

- m-selectfocus which contains procedures for determining the next focus of attention (in this case the next tracking agent to invoke).
- m-scheduler which contains procedures for scheduling the next processing action (a pair of knowledge source in m-tracking-component and an object in m-situation-board to act on).

In the process of customizing m-selectfocus the designer chooses a dynamic focus selection strategy. KASE does not possess enough knowledge to completely synthesize a dynamic focus selection algorithm. So, it simply records this decision and informs the user that he needs to provide an algorithm that takes as input a set of available tracking agents and returns the most promising one. KASE also maintains a record of modules that have not been fully customized.

At this point, the designer decides to shift his attention to m-tracking-component module (realizing, opportunistically, that he first needs to determine the set of knowledge sources in the m-tracking-component module before beginning to design the dynamic focus selection algorithm). Recent studies (15) 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

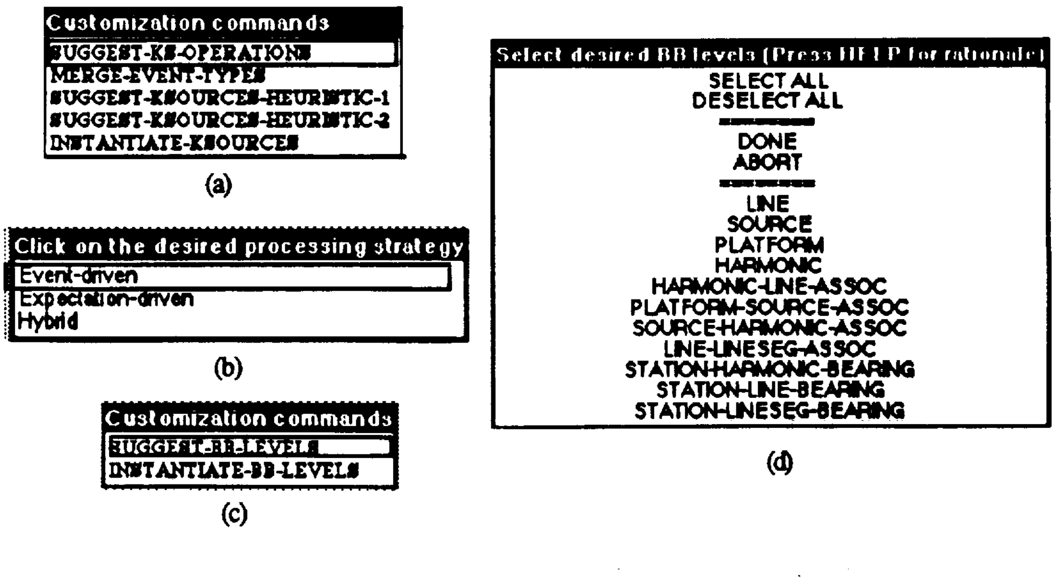


Figure 8. Customizing modules in the generic architecture

navigate among different components of the design (6).

5.3 m-Tracking-Component

The m-Tracking-Component module contains as parameters a set of submodules called tracking agents (also called ksources) where each tracking agent has a set of operations (called ks-operations) associated with it. Each operation in a tracking agent takes as input some information from m-Situation-Board and updates the information on it as a result of the operation. The tracking agents are selected by m-Scheduler based on their potential for contributing to the solution. The customization commands available for m-tracking-component is shown in Figure 8(a). The designer begins by selecting suggest-ks-operations and KASE responds with the following message:

"You need to first instantiate the BBlevels parameter of m-BBPanel module!"

In KASE, the dependencies between the various parameter values of a generic architecture must be explicitly stated (see definition of customization commands in Section 2). These are used in two ways: First, during customization, if the value of a particular parameter depends on the values of some other parameters, KASE uses these dependencies to guide the design process (as in the above case). Second, KASE maintains a history of design actions and uses the dependencies to restructure the history into a lattice; this enables KASE to localize effects of changes in the design and provide an efficient replay mechanism (7).

5.4 m-BBPanel

Guided by KASE the designer proceeds directly to the m-BBPanel module (a submodule of m-SituationBoard). The customization commands available for this module are shown in Figure 8(c), and illustrate another feature of KASE: *suggest-and-instantiate*. Until now most of KASE's customization methods were fairly straightforward – selection of an option from a set of pre-computed alternatives, or application of a set of transformations to a code template. This customization command is an example of the use of heuristic design rules. For parameter values generated by using heuristic design rules, KASE creates a *suggestions-workspace* and initially puts suggested values of the parameters in this workspace. These are meant to be default values for the parameters. The designer can examine the suggested values, ask for rationales for the suggestions, and edit them without actually committing the changes to the architecture. When satisfied, the designer can instantiate the architecture parameter with the suggested values (alternatively he may reject these values). The intention is to make the design process comprehensible to, as well as controllable by, the designer, while still retaining the ability to provide useful default solutions.

On clicking the suggest-bb-levels command from the customization menu of m-BBPanel KASE presents the list of objects and relations that should be a part of this module (Figure 8(d)). The designer can ask KASE to explain its suggestions and KASE uses annotated text templates associated with the rules to provide explanations, for example,

"Because LINE is an instance of OBJECTS-TO-BE-TRACKED and all OBJECTS-TO-BE-TRACKED must be represented on M-BBPANEL..."

The designer can use this rationale to modify his requirements and/or refine a KASE design heuristic.

5.5 m-Tracking-Component Revisited

Having instantiated the BBlevel parameter the designer returns to the m-Tracking-Component module and re-selects the suggest-ks-operations command (Figure 8 (a)). The ks-operations consist of all operations required to compute and monitor the various properties of the objects and relations to be tracked – for example, position attribute in source shown in Figure 5. The structure of the ks-operations depends on the overall solution strategy selected. Since it was decided to design the system as a pure event-driven system, the structure of a knowledge source operation consists of three components – (i) an operation that takes as input some information from the m-BBPanel module and updates m-BBPanel as a result, (ii) a set of events called *triggers* that signal that the knowledge source operation might contribute some information on the m-SituationBoard module, and (iii) a set of events called *posted-events* that represent the changes on the m-BBpanel as a result of the operation. (This is described in more detail in (5)).

KASE first determines the set of all operations that can affect any of the objects or relations stored in m-BBPanel. 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 9). For the HASP example, this results in the creation of 46 events and 26 knowledge source operations. 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 ks-operation.

There are other customization commands provided by KASE that automate some of the more frequently

Heuristic 1 (Determining types of events).

If an object is represented on the BBlevel, 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 9. Examples of some heuristics used to determine the set of events triggering ks-operations.

occurring design activities for such architectures, for example, design optimizations. One such optimizing command, shown in Figure 8(a), 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. (3, 16): *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-event* component of each *ks-operation*), 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 optimization tasks can be identified and mechanized.

There are several other customization commands provided by KASE, the details of which are not important for the purposes of this paper. We have successfully used the same set of customization commands to synthesize different variants of a generic architecture for two different domains, demonstrating the reuse of both the architecture structure as well as the design process.

6. Related Work

Work on supporting the synthesis of domain-specific software systems is recently receiving widespread attention (17). The notion of using generic architectures as a basis for providing this support is the subject of a major DARPA sponsored research initiative (the DSSA project). Our work contributes to the DSSA effort by providing a framework for building domain and architecture specific design environments.

The LEAP project (18) represents an approach that is closely related to ours: an architecture is represented as a set of reusable components which are specialized in an interactive environment using design rules. However, the domain model for an application is not explicitly represented and the design rules (corresponding to our *customization knowledge*) are acquired interactively from an end-user during design development.

Two other related works include the ARIES project (13)which is concerned with acquiring specifications in a formal language which could then be converted into an efficient implementation using transformation rules and the work on Composite System Design (19), which attempts to design composite systems from formal statements of requirements.

The general approach of creating software artifacts by knowledge-based refinement of an abstract artifact is being investigated by numerous researchers (e.g. (20, 21, 22))

7. Conclusions and Future Work

Our description of a generic software architecture embodies concepts and information from many knowledge domains. From one perspective a generic architecture is an object of the software domain in which the artifact is described using concepts such as data flow, control flow, and data type. From another perspective, it is a high-level description of a task and its solution. From yet another perspective, a generic architecture is an ontological framework within which the application domain can be modeled; that is, a generic architecture can be viewed as a *meta-model*.

Such a generic architecture when used as a basis for *reuse* provides reuse at the level of entire systems instead of at the level of algorithms or subroutines. The meta-model associated with the architecture can be used to facilitate domain modeling which currently constitutes a significant bottleneck in creating domain specific systems. Generic architectures also enable formalization of the design process which in turn leads to design with fewer errors as well as efficient maintenance and redesign. Unlike application generators where the customization knowledge is embedded in the macros and interpreters of the application generator, the design process knowledge is represented explicitly which, we believe, increases the generality and flexibility of the design environment.

Some of the current limitations in our approach include the assumption that the application shares the ontology and design of the solution assumed in the architecture. Communicating and enforcing these commitments are critical issues. In addition, acquiring architectural abstractions and the associated customization knowledge may not be easy for some classes of applications. However, abstraction needs to be done once, and for a class of application that have many potential instances we feel the advantages outweigh the disadvantages. In principle, it is also possible for a designer to modify the customization knowledge by modifying the design rules implemented in KASE. However, in practice we do not expect a designer to do that, and would like to provide tools that would automatically modify the customization process. 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 by others (8, 23) and we plan to incorporate it in KASE.

References

1. W. van Melle, A Domain Independent System that aids in Constructing Consultation Programs, PhD, Computer Science Department, Stanford University (1980).

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

3. D. R. Smith, KIDS: A Semi-automatic Program Development System, *IEEE Transactions on Software Engineering* 16, 1024-1043 (1990).

4. H. P. Nii, N. Aiello, S. Bhansali, R. Guindon, L. Peyton, Knowledge Systems Laboratory, Computer Science Department, Stanford University, Knowledge Assisted Software Engineering (KASE): An introduction and status (1991).

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

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

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

8. P. Garg, S. Bhansali, Process Programming by Hindsight, 14th International Conference on Software Engineering Melbourne, Australia, 1992),

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

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

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

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

13. W. L. Johnson, M. S. Feather, Using Evolution Transformations to Construct Specifications, in *Automating Software Design* M. Lowry, R. McCartney, Eds. (AAAI Press, Cambridge, MA, 1991).

14. H. B. Reubenstein, . C. Waters, The Requirements Apprentice: Automated Assistance for Requirements Acquisition, *IEEE Transactions on Software Engineering* 17, 226-240 (1991).

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

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

17. Notes, *AAAI Workshop on Automating Software Design* , San Jose, CA, 1992).

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

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

20. M. D. Lubars, M. T. Harandi, Addressing Software Reuse through Knowledge-based Design, in *Software Reusability* T. J. Biggerstaff, A. J. Perlis, Eds. (ACM Press, New York, New York, 1989), vol. 2, pp. 345-377.

21. N. Maiden, A. Sutcliffe, Analogical Matching for Software Reuse, 6th Annual Knowledge-Based Software Engineering Conference Syracuse, NY, 1991), pp. 101-112.

22. N. Iscoe, et al., Model-Based Software Design, AAAI Workshop on Automating Software Design San Jose, CA, 1992), pp. 72-77.

23. S. C. Bailin, R. H. Gattis, W. Truszkowski, A Learning-based Software Engineering Environment, 6th Annual Knowledge-Based Software Engineering Conference Syracuse, NY, 1991), pp. 251-263.