Engineering Software Design Processes to Guide Process Execution[*]

Xiping Song
Siemens Corporate Research Inc.
755 College Road East
Princeton, NJ 08540
song@scr.siemens.com

Leon J. Osterweil
Computer Science Department
Univ. of Massachusetts
Amherst, MA 01003
ljo@cs.umass.edu

Abstract

Using systematic development processes is an important characteristic of any mature engineering discipline. In current software practice, Software Design Methodologies (SDMs) are intended to be used to help design software more systematically. This paper explicitly shows, however, that one well-known example of such an SDM, Booch Object-Oriented Design (BOOD), as described in the literature is far too vague to provide specific guidance to designers, and is too imprecise and incomplete to be considered as a fully systematic process for specific projects.

To provide more effective and appropriate guidance and control in software design processes, we applied the process programming concept to the design process. Given two different sets of plausible process requirements, we elaborated two more detailed and precise design processes that are responsive to these requirements. We have also implemented, experimented with, and evaluated a prototype (called Debus-Booch) that supports the execution of these detailed processes.

1 Introduction

If software engineering is to make solid progress towards becoming a mature discipline, then it must move in the direction of establishing standardized, disciplined methods and processes that can be used systematically by practitioners in carrying out their routine software development tasks. We note that such standardized methods and processes should not be totally inflexible, but indeed must be tailorable and flexible to enable different practitioners to respond to what is known to be a very wide range of software development situations in correspondingly different ways. Regardless of this, however, the basis of a mature discipline of software engineering seems to us to entail being able to systematically execute a clearly defined process in carrying out these tasks. In this paper we refer to a process as being "systematic" if it provides precise and specific guidance for software practitioners to rationally carry out the routine parts of their work.

As design is perhaps the most crucial task in software development, it seems particularly crucial that software design processes be clearly defined in such a way as to be more systematic. Humphrey [Hum93] says that "one of the great misconceptions about creative work is that it is all creative. Even the most advanced research and development involves a lot of routine. ... The role of a process is to make the routine aspects of a job truly routine." We agree with this, and believe that design as a creative activity still contains a lot of routine which can be systematized. For example, making each design decision is probably creative (e.g, deciding if an entity should be a class when using an object oriented design method). However, the order of making each of these related design decisions can be relatively more systematic (e.g., identify each class first and then define its semantics and relations). We also anticipate that with progress in design communities, design methodologies will provide more routine which can be systematized. This will help adapt SDMs into practice more easily and thus improve productivity and software quality.

This paper describes our work that is aimed at this goal—namely to make SDM processes more systematic and thus more effective in guiding designers. This work begins with the assumption that the large diversity of Software Design Methodologies (SDM's) provides at least a starting point in efforts to provide the software engineering community with such well-defined and systematic design processes. This paper concentrates on the Booch Object Oriented De-

*This research is supported by the Advanced Research Projects Agency, through ARPA Order #6100, Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation. This work is also sponsored by the Advanced Research Projects Agency under Grant Number MDA972-91-J-1012.

sign Methodology (BOOD) [Boo91] in order to provide specificity and focus. The paper shows, however, that BOOD, as described and defined in the literature, is far too vague to provide specific guidance to designers, and is too imprecise and incomplete to be considered a very systematic process for the needs of specific projects. On the other hand, we did find that BOOD could be considered to be a methodological framework for a family of such processes.

Our work builds upon the basic ideas of process programming [Ost87], which suggest that software processes should be thought of as software themselves, and that software processes should be designed, coded, and executed. That being the case, we found that BOOD, as described in the literature, is far closer to the architecture, or high-level design, of a design process than to the code of such a process. As such, BOOD is seen to be amenable to a variety of detailed designs and encodings, each representing an elaboration of the BOOD architecture, and each sufficiently detailed and specific that it can be systematic in a way that is consistent with superior engineering practice in older, more established engineering disciplines.

In the remainder of this paper we indicate how and why we believe that BOOD should be considered a software design process architecture. We then suggest two significantly different detailed designs that can be elaborated from BOOD, each of which can be viewed as a more detailed elaboration upon the basic BOOD architecture. We show that these elaborations can be defined very precisely through the use of such accepted software design representations as OMT [RBP+91], and through the use of process coding languages such as APPL/A. Indeed, this paper shows that the use of such formalisms is exactly what is needed in order to render these elaborations sufficiently complete and precise that they can be considered to be systematic.

Thus, the paper indicates a path that needs to be traveled in order to take the work of software design methodologists and render it the adequate basis for a software engineering discipline.

First (in Section 2), we define the process architecture provided by BOOD, and then describe two processes elaborated from the architecture. Second (in Section 3), we describe a prototype that supports designers in carrying out the execution of these processes, illustrating how these differently elaborated processes support different execution requirements. Third (in Section 4), we describe our experience of using the prototype and summarize some of the main issues that have arisen in our efforts to take the design process architectures that are described in the literature to the level of encoded, systematic design processes.

2 The BOOD Architecture and Two Elaborations

2.1 Overview of BOOD

We decided to experiment with BOOD because BOOD is widely used, and provides a few application examples that are very useful in helping us to identify the key issues in elaborating BOOD to the level of executable, systematic processes. A detailed description of BOOD can be found in [Boo91]. In this section, we present only a brief description of the architecture of the BOOD process. We believe that it can be summarized as consisting of the following steps:

1. **Identify Classes/Objects:** Designers must first analyze the application requirements specification to identify the most basic classes and objects, which could be entities in the problem domain or mechanisms needed to support the requirement. This step produces a set of candidate classes and objects.

2. **Determine Semantics of Classes/Objects:** Designers must next determine which of the candidate classes should actually be defined in the design specification. If a class is to be defined, designers will determine its semantics, specifying its fields and operations.

3. **Define Relations among Classes:** This step is an extension of step 2. Designers must now define the relationships among classes, which include use, inheritance, instantiation and meta relationships. Steps 2 and 3 produce a set of class and object diagrams and templates, which might be grouped into class categories.

4. **Implement Classes:** Designers must finally select and then use certain mechanisms or programming constructs to implement the classes/objects. This step produces a set of modules, which might be grouped into subsystems.

BOOD provides more hints and guidelines on how to carry out these steps. However, BOOD provides no further explicit elaboration on the details of these steps. Thus designers are left to fill in important details of how these complex, major activities are to be done. As a result, there is a considerable range of variation and success in carrying out BOOD. Further, the process carried out by those who are relatively more successful is not documented, defined or described in a way that helps them to repeat it effectively, or for them to pass on so that others can reuse it. We believe that this is the sense in which BOOD, as described in the literature, is a process architecture. It provides the broad features and outlines of how to produce a design. It supplies elements that can be thought of and used as building blocks for specific approaches to design creation. On the other hand, it provides no specific guidance, details or procedures. These are to be filled in by others who, we claim, then become design process designers (e.g., the authors of [HKK93]) and implementors when the method is applied to specific projects or organizations.

2.2 Process Definition Formalism

Earlier experiences [KH88, CKO92] have shown that the State-Charts formalism [HLN+90] is a powerful vehicle for modeling software processes. Thus, we use a variant of State-Charts [HLN+90], the dynamic modeling notation of the Object Modeling Technique (OMT) [RBP+91], to model the processes that we will elaborate from BOOD. As shown later, we believe these dynamic models of BOOD processes are sufficient to demonstrate our point

Generally, our approach is to use the notion of a *state* (denoted as a labeled rounded box) to represent a step of a BOOD process, the notion of an *activity* (the text inside a rounded box and after "Do:") to represent a step which does not contain any other steps. According to OMT, the order for performing these activities can be sequential, parallel or some other forms. We use the order in which the activities is listed to recommend a plausible order for performing those activities. A *transition* (denoted as a solid arc) denotes moving from one design step to another. The text labels on a transition denote the *events* which cause the transition. The text within brackets indicates *guarding conditions* for this transition. The text within parentheses denotes attributes passed along with the transition. A state could have sub-states, each of which denotes a sub-step of the step.

Indeed, a modelling formalism is generally inadequate for characterizing certain details of processes. We found that sometimes it was necessary to specify these details in order to render the process we were attempting to specify sufficiently precisely that it could realistically be considered to be systematic. For example, OMT does not provide a capability for specifying the sequencing of two events which are sent by the same transition. Specification of this order might well be the basis for important guidance to a designer about which design issues ought to be considered before which others. Thus, we found it necessary to supplement OMT, by using a process coding language called APPL/A [SHO90b] to model such details. APPL/A is a superset of Ada that supports many features that we found to be useful. Some examples of APPL/A code are also provided in subsequent sections of this paper.

Note that the goal of this work is to use these process models and codes to demonstrate the diversity and details of the processes that can be elaborated from an SDM. As shown later, these dynamic models of BOOD processes are sufficient to demonstrate this point. Thus, we did not develop OMT's object models and function models for BOOD processes.

2.3 Modeling the BOOD Architecture

Fig. 1 represents an OMT model of the original BOOD process architecture, as described in [Boo91]. In this architecture, we merged step 2 and 3 of the original BOOD process because our experience shows that it is hard to separate those steps in practice (Booch himself also considers that step 3 is an extension of step 2 [Boo91]). We believe that this model is considerably more precise than the informal description originally provided. It is still quite vague and imprecise on many important issues, however. Booch [Boo91] claims that this vagueness is necessary in order to assure that users will be able to tailor and modify it as dictated by the specifics of particular design situations. For example, step 2 of Fig. 1 does not define the order for editing various BOOD diagrams and templates. It does not define clearly which of the diagrams or templates must be specified in order to move from step 2 to step 3. Booch claims that different designers might have important and legitimate needs to elaborate these details in different ways (Chapters 8-12 of [Boo91] provide a few examples).

We found that there are indeed many ways in which these details might be elaborated precisely and that many of these different variants might offer better guidance. The differences might well arise from differences in application, differences in organization, differences in personnel expertise, and differences in the nature of specific project constraints. Once these differences have been understood and analyzed, however, the design process to be carried out should be defined with suitable precision. Such precise definitions are needed in order to support adequate improvements of the efforts of novices. In addition, we believe that there are expert designers who have internalized very

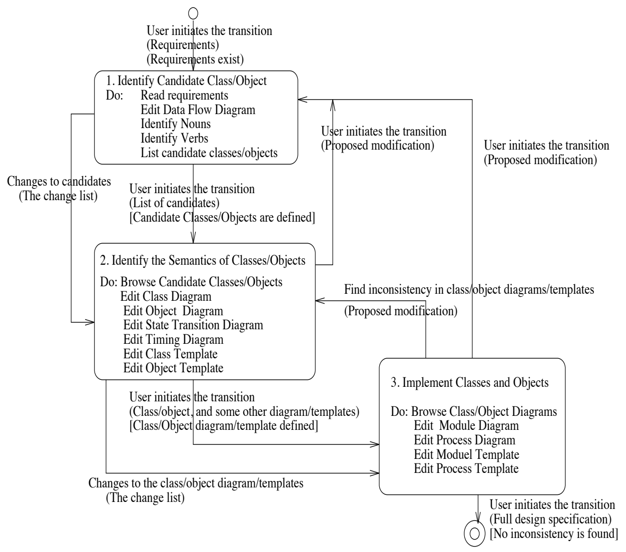

User initiates the transition
(Requirements)
(Requirements exist)
1. Identify Candidate Class/Object
Do: Read requirements
Edit Data Flow Diagram
Identify Nouns
Identify Verbs
List candidate classes/objects
User initiates the transition
(Proposed modification)
User initiates the transition
(Proposed modification)
Changes to candidates
(The change list)
User initiates the transition
(List of candidates)
[Candidate Classes/Objects are defined]
2. Identify the Semantics of Classes/Objects
Do: Browse Candidate Classes/Objects
Edit Class Diagram
Edit Object Diagram
Edit State Transition Diagram
Edit Timing Diagram
Edit Class Template
Edit Object Template
Find inconsistency in class/object diagrams/templates
(Proposed modification)
3. Implement Classes and Objects
Do: Browse Class/Object Diagrams
Edit Module Diagram
Edit Process Diagram
Edit Moduel Template
Edit Process Template
User initiates the transition
(Class/object, and some other diagram/templates)
[Class/Object diagram/template defined]
Changes to the class/object diagram/templates
(The change list)
User initiates the transition
(Full design specification)
[No inconsistency is found]

Figure 1: A Process Architecture of BOOD

specific and very effective elaborations of the BOOD architecture and that the more these are defined precisely, the more important design expertise may be understood, reused, automated, and improved.

In order to make the above remarks specific, we now discuss two possible elaborations of the BOOD architecture. In addition, using these examples we can show the need for, and power of, both design and code representations as vehicles for making design processes clear and thereby providing more effective guidance.

2.4 Two Examples of BOOD Process Refinements

2.4.1 Examples of Software Project Types

First, we characterize two different types of projects for which we will elaborate variants of design processes, within the outlines of the BOOD architecture (see Project Properties columns in Table 1). The parameters of these characterizations are 1) implementation language, 2) documentation requirements, 3) project schedule, 4) designer skill, 5) software operation domain, 6) software domain, and 7) maturity of the software domain.

Based upon our experiences, we identified these two project types as representatives of projects commonly encounted in software engineering practice (see Table 1). For example, an instance of project type 1 could be a defense-related or a medical systems project while an instance of project type 2 could be a civilian project. We expected that elaborating processes to fit the requirements of these two different types of projects would help us understand the range of processes that could be elaborated from BOOD.

The seven characterization parameters were chosen because our earlier work indicated that these parameters are likely to have major and interesting effects upon design process elaboration. For example, when consulting with Siemens medical companies, we found that the U.S. Food and Drug Administration (FDA). has specific documentation requirements, and requires control and monitoring of corrective actions on the product design. [FDA89] says that "when corrective action is required, the action should be appropriately monitored... Schedule should be established for completing corrective action. Quick fixes should be prohibited." This certainly affects how an SDM should be applied to a specific project.

The application examples described in [Boo91] also provide us with some details that seemed likely to be useful in employing these parameters to help us to derive these BOOD-based design processes. For example, one of Booch's examples indicates that if C++ is to be the eventual application coding language, then class/object diagrams would not need to be translated into module diagrams. In addition, Booch's problem report application example [Boo91] helps us to understand the process requirements for developing an information processing system. For instance, that example shows that the method must be tailored to support the design of database schemas. His traffic control example helps us to understand the process requirements for developing a large scale, device-embedded system.

2.4.2 The Processes Elaborated from BOOD

In this section we present portions of the OMT diagrams used to define details of each of these two elaborations on the basic BOOD architecture. We then further refine parts of them down to the level of executable code. Each of these processes is clearly a "Booch Design Process", each represents what we consider to be a completely plausible design process, and each is quite completely and precisely defined—to the point of being systematic for the specific kinds of projects. These two processes demonstrate the point that there is a great deal of imprecision in the current definition of "Booch Object-Oriented Design." They also indicate how BOOD can be elaborated, and what the range of elaboration might be when it is applied to specific projects.

We will refer to our first elaborated process as *the Template Oriented Process (TOP).* It emphasizes defining various BOOD templates (e.g., the class template) as it hypothesizes the importance of carrying

1. A Template Oriented Example		2. A Diagram Oriented Example	
Project Properties	**Process Requirements**	**Project Properties**	**Process Requirements**
Must be coded in Ada	Specify Module Diagram	Must be coded in C++	Guide designers not to specify Module Diagram since it is not needed in this case.
Must incorporate very complete documentation	Requires specification of all templates	Only minimum documentation required	No need to enforce specifying all templates
Long-term	Allow full documentation	Short-term	Encourage use of existing code
Skilled design team	Less process guidance More process flexibility	Inexperienced design team	More process guidance Less process flexibility
Safety-critical (e.g., Medical Systems)	More change control needed to satisfy FDA's requirements	Non safety-critical	Less change control needed
Large scale, device-embedded system	Use structured analysis Support partitioning domain	Information processing system	Single, familiar domain. Need to support schema design
State of the art project	Need to support prototyping	Well-understood	No support for prototyping needed. Need to support code reuse

Table 1: Project Characteristics and Process Requirements

out a design activity that delivers very complete documentation. The TOP's emphasis on complete documentation can be seen by noting that we have refined steps 2 and 3 of Fig 1 into the more detailed model defined in Fig. 2.

We further hypothesized in designing the TOP that the software to be developed is to be safety-critical, and that, therefore, the TOP should enforce more control over design change as this is often required by government agencies to ensure product quality. Accordingly note that the high level design of the TOP incorporates an approval cycle for all changes to previously defined artifacts.

On the other hand, we hypothesized that the TOP is to be executed by skilled and experienced designers. Because of this, we did not refine the detailed design activities into lower level steps. Our expectation here is that such designers would insist upon freedom and flexibility that this would be given them. This also illustrates that it is possible to define a design process precisely, yet still provide considerable freedom and flexibility to practitioners. In addition we designed the TOP to allow for a certain degree of flexibility in making transitions from one step to another. We have also included the possibility of incorporating a prototyping subprocess into this process.

We refer to our second elaborated process as *the Diagram Oriented Process (DOP)*, as it emphasizes specifying BOOD diagrams. We derived this process from Booch's Home Heating System example [Boo91]. In the DOP we hypothesized that there are only weak requirements in the area of documentation, and we, therefore, do not design in the need for designers to specify BOOD's templates (see Figures 4). We also hypothesized that the product being designed will be coded in a language that provides direct support for programming classes and objects. For this reason, the DOP omits step 3 of the general model shown in Fig. 1 as part of its elaboration, leaving the model defined in Fig. 4. Note that this elaboration incorporates fewer top-level steps than the general BOOD model does.

We also hypothesized that the DOP is aimed at supporting novice designers, and so the DOP provides detailed guidelines for identifying classes/objects (see Figures 5, 6, and 7). In addition, the DOP assumes that a great deal of importance is placed upon reuse. In response, the DOP incorporates steps that guide designers to reuse existing software components (see Fig. 7).

The job of creating more specific and detailed elaborations of BOOD is not limited solely to modification of the processing steps of BOOD. It also entails specifying the flow of control between these steps and their substeps. A good example of the importance of these specifications can be seen by examining how change management is handled in these design processes.

We use the term *forward change management* to denote a transition used to maintain consistency between a changed artifact and its dependent artifacts, that are normally specified at a later stage of the process. For example, a designer may add a class to a candidate class list (in step 1 of Fig. 2) . This results in forcing designers to redo step 2 to consider adding a corresponding class to the class diagram. There is virtually no guidance in BOOD about precisely how this is to be done, or how the critical and tricky issues of consistency management are to be addressed. Thus there is a clear need for more detailed guidance on automatic change control. One way this can be done is to refine this high-level transition further as shown in Fig. 8. In Fig. 8, a dotted line from a transition to a class represents an event sent by the transition.

For example, the transition from Selected Class A to Rejected Class A, which is caused by updating candidate class A's field Needed to FALSE (i.e. class A is no longer needed), sends event Delete Class A to class Class. Clearly this refinement is simply one of a very large assortment of possible refinements. We do not claim that it is the only one or the "right" one. We do claim, however, that supplying details such as these provide specific guidance that is important for designers—especially for novice designers and for large design teams. Should it turn out that such a specifically designed process is shown to be particularly useful and desirable, then the detailed specification will also render it more amendable to computer support.

We should also note that we did not stop at the level of design diagrams in refining the meaning of forward change management, but that we went further and defined it as actual executable process code. Our code was written in the APPL/A process coding language. Fig. 9 shows the APPL/A code for the process defined in Fig. 8. Note that this code provides even more details. For example, note that this code specifies that changing a candidate class to a candidate object will cause an ordered sequence of events: 1) the insertion of an object template, 2) the removal of the class template and 3) the forwarding of that template to step 3 for editing of the object template. Again, we stress that these specific details are not to be considered the only feasible elaboration of BOOD—only one possible elaboration. We do believe, however, that in specifying the design process to this level of detail deeper understandings result, and the process becomes more systematic. In addition, by reducing the process to executable APPL/A code, it becomes possible to use the computer to provide a great deal of automated support (e.g., some types of automatic updating and consistency maintenance) to human designers.

Another kind of control flow in BOOD is *backward change management*, which is aimed at maintaining consistency between a specified artifact and all the artifacts upon which the specified artifact should depend. These artifacts are normally defined at earlier stages of the process. For example, in step 2 of Fig. 2, designers may need to define a class in a class diagram and find that this class does not correspond to any candidate class because of an incomplete or faulty analysis of the application requirements. Thus, designers have to go back to earlier steps, reviewing the requirements and possibly redoing step 1 to add this class to the candidate class list. This transition can be refined and coded in a manner similar to what was described in the case of forward change management.

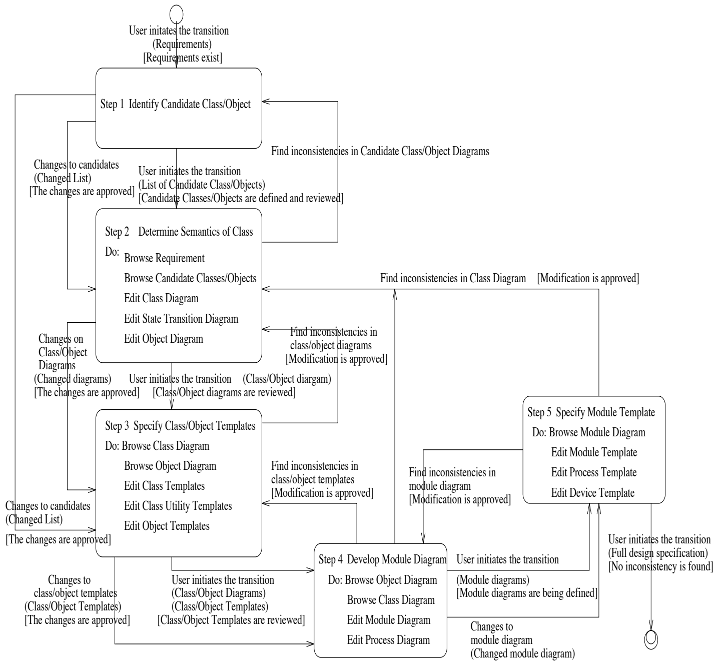

User initates the transition
(Requirements)
[Requirements exist]
Step 1 Identify Candidate Class/Object
Find inconsistencies in Candidate Class/Object Diagrams
Changes to candidates
(Changed List)
[The changes are approved]
User initiates the transition
(List of Candidate Class/Objects)
[Candidate Classes/Objects are defined and reviewed]
Step 2 Determine Semantics of Class
Do: Browse Requirement
Browse Candidate Classes/Objects
Edit Class Diagram
Edit State Transition Diagram
Edit Object Diagram
Find inconsistencies in Class Diagram [Modification is approved]
Find inconsistencies in
class/object diagrams
[Modification is approved]
Changes on
Class/Object
Diagrams
(Changed diagrams)
[The changes are approved]
User initiates the transition (Class/Object diargam)
[Class/Object diagrams are reviewed]
Step 3 Specify Class/Object Templates
Do: Browse Class Diagram
Browse Object Diagram
Edit Class Templates
Edit Class Utility Templates
Edit Object Templates
Find inconsistencies in
class/object templates
[Modification is approved]
Changes to candidates
(Changed List)
[The changes are approved]
Step 5 Specify Module Template
Do: Browse Module Diagram
Edit Module Template
Edit Process Template
Edit Device Template
Find inconsistencies in
module diagram
[Modification is approved]
Step 4 Develop Module Diagram
Do: Browse Object Diagram
Browse Class Diagram
Edit Module Diagram
Edit Process Diagram
User initiates the transition
(Module diagrams)
[Module diagrams are being defined]
User initiates the transition
(Full design specification)
[No inconsistency is found]
Changes to
class/object templates
(Class/Object Templates)
[The changes are approved]
User initiates the transition
(Class/Object Diagrams)
(Class/Object Templates)
[Class/Object Templates are reviewed]
Changes to
module diagram
(Changed module diagram)

Figure 2: Top-level Process Definition of the Template Oriented Process

These process definitions, including both main flow and change management transitions, explicitly and clearly demonstrate how the published Booch Object Oriented Design description can be elaborated into a precisely defined process to provide more effective guidance for specific projects. Our research indicates that this observation is quite generally applicable to the range of SDM's that are currently being espoused widely in the community. There are a number of reasons for this imprecision. We have already noted that the imprecision is there intentionally to permit wide variation in design processes to match similarly wide design process contexts and requirements. While we neither doubt nor dispute this need, we believe that our work has shown that it can be met more effectively through tailoring SDMs for specific needs of projects. These processes resulting from the tailoring, and supported by the appropriate tools, provide more effective guidance and help implement various recommended practices (e.g, those recommended by FDA [FDA89]). In the next sections, we discuss how to support the execution of the elaborations of the BOOD architecture that we have just described.

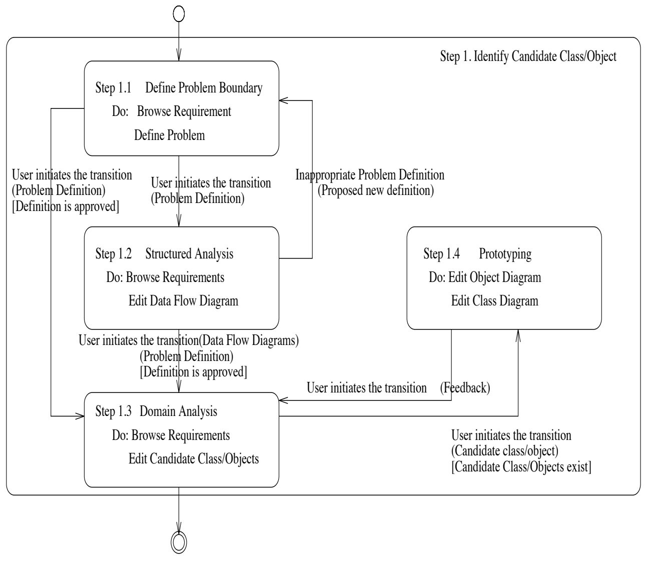

Step 1. Identify Candidate Class/Object
Step 1.1 Define Problem Boundary
Do: Browse Requirement
Define Problem
User initiates the transition (Problem Definition) [Definition is approved]
User initiates the transition (Problem Definition)
Inappropriate Problem Definition (Proposed new definition)
Step 1.2 Structured Analysis
Do: Browse Requirements
Edit Data Flow Diagram
Step 1.4 Prototyping
Do: Edit Object Diagram
Edit Class Diagram
User initiates the transition(Data Flow Diagrams) (Problem Definition) [Definition is approved]
User initiates the transition (Feedback)
Step 1.3 Domain Analysis
Do: Browse Requirements
Edit Candidate Class/Objects
User initiates the transition (Candidate class/object) [Candidate Class/Objects exist]

Figure 3: Second-level Process Definition of Template Oriented Process: Refinement of Step 1

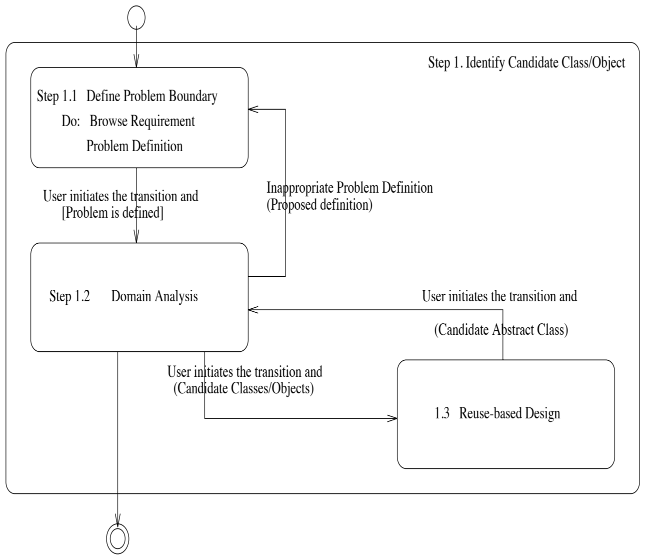

Step 1. Identify Candidate Class/Object
Step 1.1 Define Problem Boundary
Do: Browse Requirement
Problem Definition
User initiates the transition and [Problem is defined]
Inappropriate Problem Definition (Proposed definition)
Step 1.2 Domain Analysis
User initiates the transition and
(Candidate Abstract Class)
User initiates the transition and (Candidate Classes/Objects)
1.3 Reuse-based Design

Figure 5: Second-level Process Definition of Diagram Oriented Process

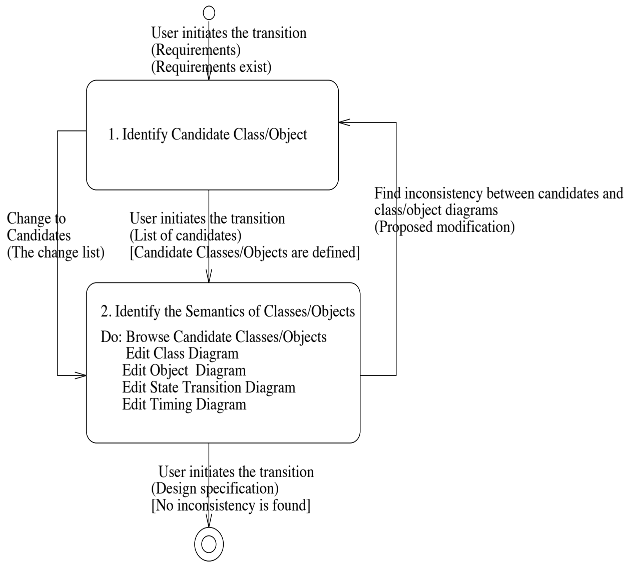

User initiates the transition (Requirements) (Requirements exist)
1. Identify Candidate Class/Object
Change to Candidates (The change list)
User initiates the transition (List of candidates) [Candidate Classes/Objects are defined]
Find inconsistency between candidates and class/object diagrams (Proposed modification)
2. Identify the Semantics of Classes/Objects
Do: Browse Candidate Classes/Objects
Edit Class Diagram
Edit Object Diagram
Edit State Transition Diagram
Edit Timing Diagram
User initiates the transition (Design specification) [No inconsistency is found]

Figure 4: Top-level Process Definition of Diagram Oriented Process

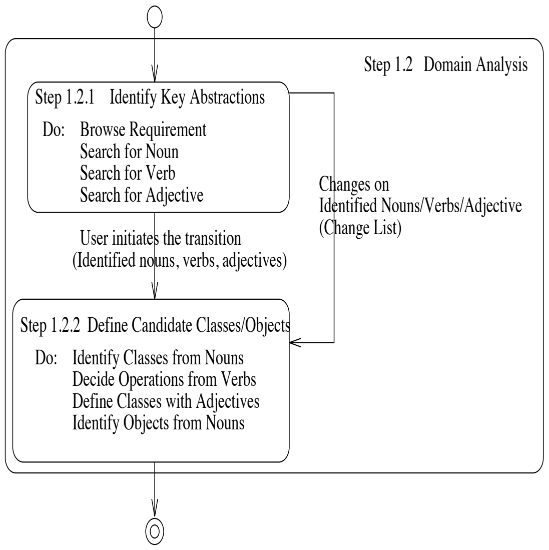

Step 1.2 Domain Analysis
Step 1.2.1 Identify Key Abstractions
Do: Browse Requirement
Search for Noun
Search for Verb
Search for Adjective
Changes on Identified Nouns/Verbs/Adjective (Change List)
User initiates the transition (Identified nouns, verbs, adjectives)
Step 1.2.2 Define Candidate Classes/Objects
Do: Identify Classes from Nouns
Decide Operations from Verbs
Define Classes with Adjectives
Identify Objects from Nouns

Figure 6: Third-level Process Definition of Diagram Oriented Process : Refinement of Domain Analysis

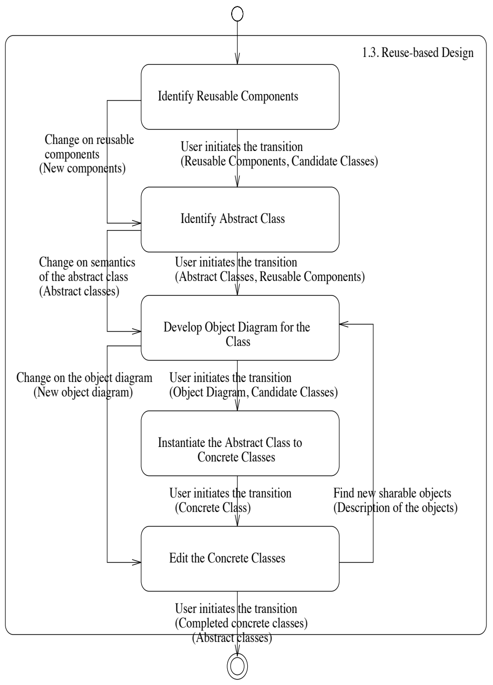

1.3. Reuse-based Design
Identify Reusable Components
Change on reusable components (New components)
User initiates the transition (Reusable Components, Candidate Classes)
Identify Abstract Class
Change on semantics of the abstract class (Abstract classes)
User initiates the transition (Abstract Classes, Reusable Components)
Develop Object Diagram for the Class
Change on the object diagram (New object diagram)
User initiates the transition (Object Diagram, Candidate Classes)
Instantiate the Abstract Class to Concrete Classes
User initiates the transition (Concrete Class)
Find new sharable objects (Description of the objects)
Edit the Concrete Classes
User initiates the transition (Completed concrete classes) (Abstract classes)

Figure 7: Third-level Process Definition of the Diagram Oriented Process : Refinement of Reuse-based Design, which is based on Booch's Home Heating System example

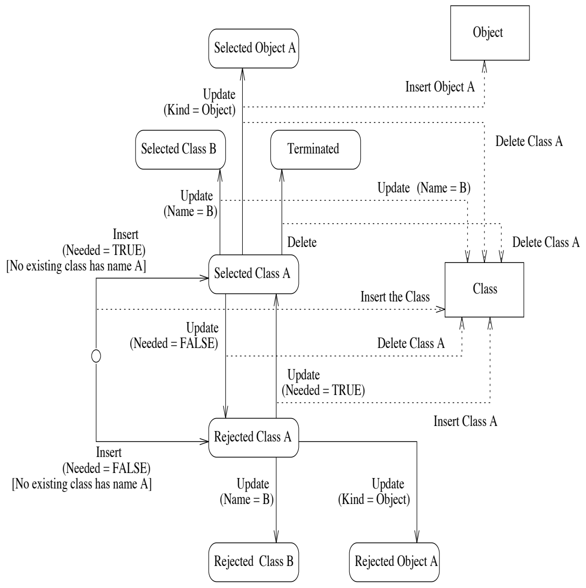


Figure 8: A Refinement of Forward Change Management: illustrates more precisely how a change in the candidate list might affect the class diagram/templates. A candidate is recorded with three fields: name, needed (indicating if it is selected as an candidate), and kind (indicating that it is a class or object)

3 Support for Executing BOOD Processes

To experiment with our ideas and demonstrate how these processes should be supported appropriately, we have developed a research prototype, called **Debus-Booch**, to support the execution of design processes of the sort that have just been described. Execution of such processes is possible as a result of their encoding in APPL/A, a superset of Ada that can be translated into Ada, and then compiled into executable code.

We note that BOOD addresses only issues concerned with supporting single users working on a single design project. As most designers must work in teams, and are often engaged in multiple projects simultaneously, a practical system for support of such users must do more than simply execute straightforward encodings of BOOD elaborations. Our Debus-Booch prototype adapts an architecture used in a previous research prototype (Rebus [SHDH$^+$91]). The architecture lets developers post (to be done) and submit (finished) tasks to a *whiteboard* to coordinate their task assignments. Since this work has been published and is not directly related to the topic of this paper, we will not describe it here.

In addition, there are a variety of difficult user interface issues to be faced in implementing a system such as this. Exhaustive treatment of all of these issues is well beyond the scope and limitations of this paper. An indication of our approaches to these and related problems can be seen from the following brief implementation discussion.

3.1 System Overview

Debus-Booch provides four levels of process guidance and support to its end-users (see Fig. 11 for their user interface representations):

1. **Process Selection (Accessed through a Console/Driver):** This enables users to select any of a range of elaborations of the BOOD architecture, or any non-atomic step of any such elaboration (as shown in Fig. 10). This is done by selecting a driver to perform a constrained sequence of steps at a certain level of the selected process step hierarchy. Debus-Booch helps users with this selection by furnishing users with access to information about the nature of these various processes and steps.

```
with candidate_rel, class_template;
trigger maintain_candidate;
  --| maintain the product of step 1
trigger body maintain_candidate is
begin
 loop
  trigger select
    upon candidate_rel.update
              (name        : string;
               needed      : boolean;
               kind        : candidate_type;
               update_name : boolean;
                  new_name : string;
             update_needed : boolean;
                new_needed : boolean;
               update_kind : boolean;
                  new_kind : candidate_type)
        completion do
          if needed = TRUE or update_needed = TRUE then
        --| change management is necessary only when
        --| candidate is selected or being updated.
        case kind is
          when class =>
            if (update_needed = TRUE) then
             if new_needed = FALSE then
             --| the candidate is no longer needed
              class_template.delete(name => name);
             else
             --| the candidate becomes needed
              class_template.insert (name => name, ...);
             end if;
             query (pname, plen, sname, slen);
             define_class_proc(pname,plen,sname, slen);
            elsif (update_name = TRUE)
              class_template.update (name => name,
                         update_name => TRUE,
                         new_name => new_name);
            end if;
            if (update_kind = TRUE and
              ((needed = TRUE and update_needed = FALSE)
              or (new_needed = TRUE and
              update_needed = TRUE)) then
             case new_kind is
              when object =>
               object_rel.insert (name => name);
               class_template.delete (name);
               define_object_proc(pname,plen,sname,slen);
              when operation => ......
             end case;
            end if;
          when abstract_class => ....
        end case
    end upon
 or
    ......
 end select
 end loop;
end maintain_candidate;
```

Figure 9: APPL/A code for defining Forward Change Management between candidate class list and class diagram/template definitions

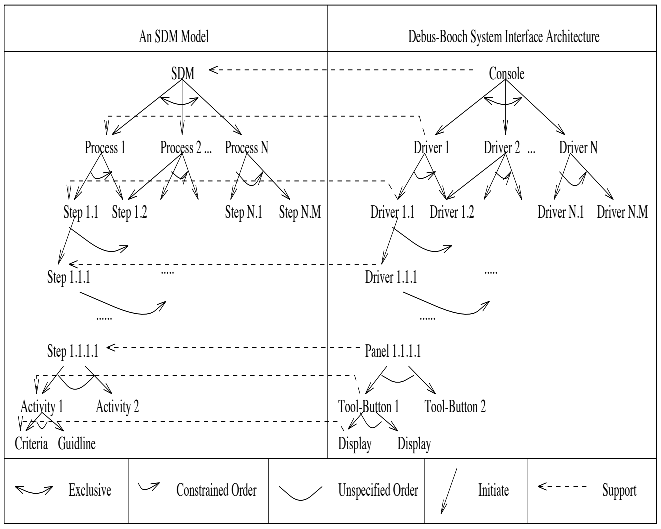


Figure 10: An SDM definition and support model

2. **Process Step Execution (Accessed through a Panel):** The user can obtain support for the sequencing and coordination of the driver activities to be performed in an elaborated design process. These activities can be divided into two categories: *required* and *optional* activities. For example, in the step used to determine the semantics of classes, designers must use Class Diagram Editor, which therefore supports a required activity; in the same step, designers *may* use a requirements browser, which therefore supports an optional activity. Designers can invoke all the tools that support the required activities by clicking on the Set Environment button. In using this access method, we help designers to set up a design environment more easily. Note that different processes may have different required activities. For example, in the template oriented process (TOP), editing the class template is a required activity. However, in contrast, using the diagram oriented process (DOP), the user cannot even access this editor.

3. **Atomic activity/support (Accessed through a Tool-Button):** The user can obtain support for a specific activity in an atomic step. For example, the user can request access to a Class Diagram Editor in order to obtain support for defining a class diagram, which is an activity performed in determining the semantics of classes.

4. **Documenation and Help Support (Accessed through Displays):** This support can be obtained in conjunction with the use of tools that support atomic activities. The displays that are made available convey a variety of information, such as the criteria, guidelines, examples, and measures [SO92] to be used to help designers understand how to carry out the activity.

Debus-Booch provides the flexibility that is needed for experienced designers. Designers can use a console display to access all of the supports listed above. For example, a designer can click on the Console's Steps button to execute any step of any elaborated BOOD process (as long as the guarding condition for this step is satisfied, otherwise, the invocation will be rejected).

Figure 10 shows how these four types of support are made available to the designers who use Debus-Booch. In particular the figure indicates the degrees of interactions that are allowed among the supports for processes, steps, and activities. In particular, note that support for process execution will be provided on an exclusive basis only, as we believe it is reasonable to use only one process at a time to design any given system, or any major part of a system. Similarly, there are constraints on furnishing support for the simultaneous execution of process steps. This is because there are often data dependencies between steps. On the other hand, support for simultaneous execution of activities is unconstrained as many design process activities must often be highly cooperative in practice. Some sets of activities must indeed be carried out in constrained orders. In this case it is necessary to group them into composite steps. The decisions about allowable degrees of concurrency were made based on our observations of the nature and structure of the process models defined in Section 2.4.2.

3.2 Scenario for Use of Debus-Booch

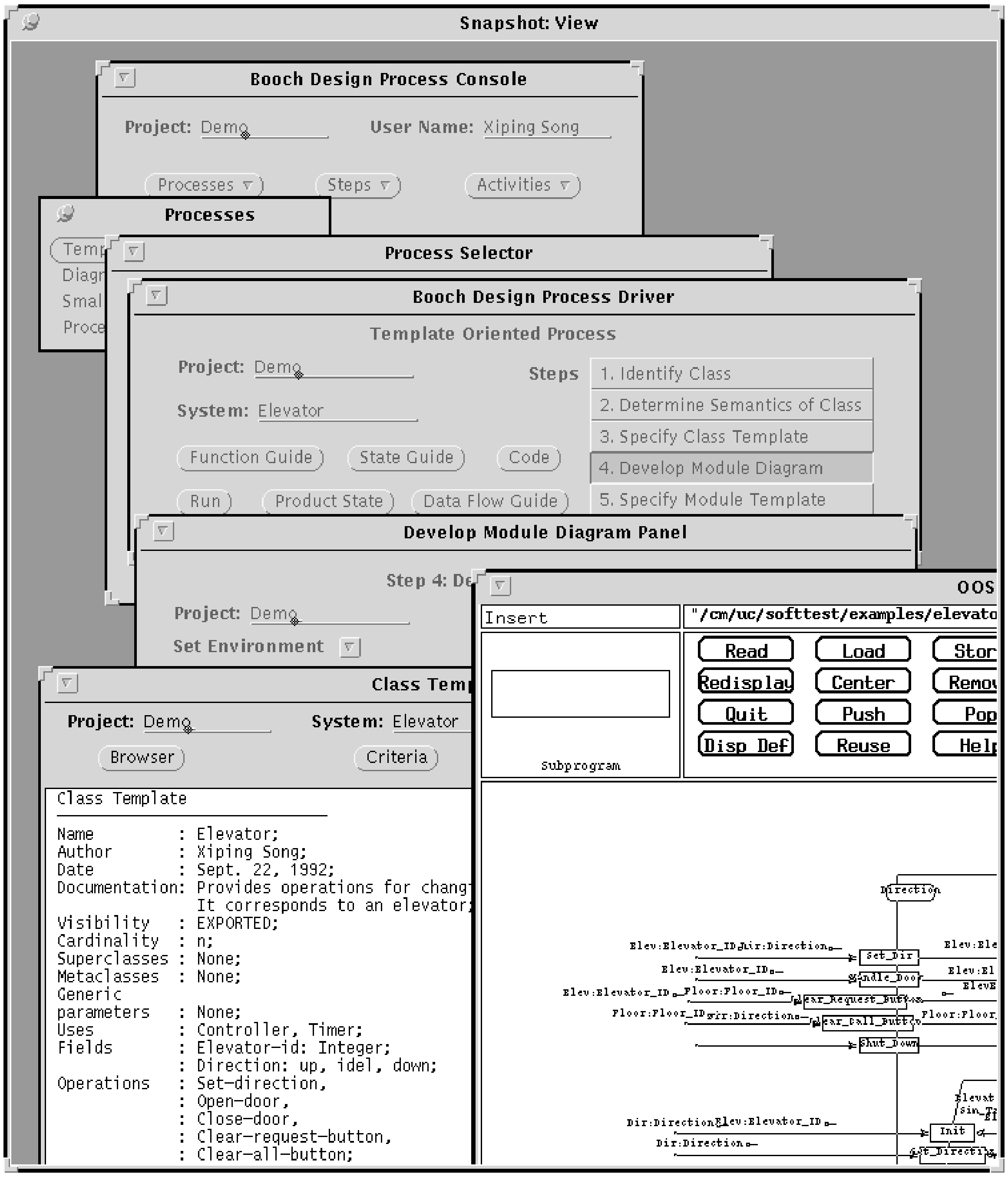
Here is a general scenario, which indicates how designers might use Debus-Booch (see Fig. 11):

1. Designers select a specific elaborated BOOD process from the menu popped up after pressing the Process button. They may select Process Selector to retrieve information about these processes. For each process, the Process Selector describes the most appropriate situations (e.g., the documentation requirements, project deadline) under which the process should be used.
2. Upon clicking on the menu item (i.e. a selected process), the corresponding driver will be initiated. Then, designers must enter the name of the subsystem to be designed. This subsystem can be assigned to them from a management process or a high-level system decomposition process (e.g., in our case, it is on the whiteboard [SHDH+91]).
3. When the subsystem name has been entered, the driver will check what design steps have been performed on this subsystem, and then automatically set the current sub-step in order to continue with the design of this subsystem. (This is tantamount to the process of restarting a suspended execution of the process from a previously stored checkpoint.) Then, the designer can click the Run button to invoke the corresponding sub-driver or atomic step support.
4. If a sub-driver is initiated, step 3 will be repeated except designers will not need to enter the subsystem name again.
5. If atomic step support is invoked, a panel appears and designers can click on its tool-buttons to invoke the tools to support the activities that should be carried out in this atomic step.
6. Having finished this step, designers can click on the next step using the Steps buttons of the driver to move the process forward. If the guarding condition (e.g., see Fig. 2) for the next step is true, the move will succeed, otherwise, the move will be rejected. After finishing the final step in the elaborated process, the designer may go back to the first step to start another iteration on the same subsystem, reviewing and revising the artifacts produced in the previous iteration. Thus, Debus-Booch also provides supports for process iteration.

As this scenario illustrates, Debus-Booch provides different supports for users who are using different process elaborations. For example, using the template oriented process, the user will be guided by the driver, (with enforcement provided by the guarding condition), to specify the module diagram as is useful when Ada is used as the implementation language. In contrast, using the diagram oriented process, the user will be directed to *not* define the module diagram as it is not considered to be of value when an object-oriented language is used.

4 Experience and Evaluation

In the past year, we have carried out two experiments and one evaluation with Debus-Booch. In the



Project: Demo   User Name: Xiping Song

Temp
Diagr
Smal
Proce

Project: Demo   Steps
1. Identify Class
2. Determine Semantics of Class
3. Specify Class Template
4. Develop Module Diagram
5. Specify Module Template

System: Elevator

Step 4: De
Project: Demo

Project: Demo   System: Elevator

"/cm/uc/softtest/examples/elevato

```
Class Template
____________________________

Name         : Elevator;
Author       : Xiping Song;
Date         : Sept. 22, 1992;
Documentation: Provides operations for chang
               It corresponds to an elevator;
Visibility   : EXPORTED;
Cardinality  : n;
Superclasses : None;
Metaclasses  : None;
Generic
parameters   : None;
Uses         : Controller, Timer;
Fields       : Elevator-id: Integer;
             : Direction: up, idel, down;
Operations   : Set-direction,
             : Open-door,
             : Close-door,
             : Clear-request-button,
             : Clear-all-button;
```

Figure 11: A Stack of Debus-Booch Windows Supporting the Booch Method

first experiment, we used the prototype to develop a design example: an elevator control system. This is a real-time system that controls the moving of elevators in response to requests of users [RC92]. It was used as an example for demonstrating how the Arcadia consortium supports the whole software development lifecycle. The system requires full documentation, and is to be implemented in Ada. It is safety-critical and device-embedded. The design team was to include the lead author and students who had finished the software design course. Thus, this project has most of the characteristics described in the Template Oriented Example (see Table 1).

Our experience with this experiment shows that the Template Oriented Process (TOP) supported our design development quite effectively. The process represented through the drivers and panels guided us to define the BOOD templates and the module diagrams. For example, the designers were guided to define the problem boundary first and then identify candidate classes such as Controller, Button, Floor, and Door. In this experiment, we found that the Set-Environment button was most frequently used and was effective in guiding designers to define those required diagrams and templates. The flexibility offered by the process allowed the designers to modify some intermediate design specifications. For example, the designers often moved back to Step 1 from Step 2 (i.e., the Determine Semantics of Class step of Fig. 2) to modify the candidate classes. However, to ensure system safety, this process enforced stricter control over the other backward changes which directly affect the actual design documentation. For example, the transition from Step 3 to Step 2 of Fig. 2 was more strictly monitored. In using the prototype, we found the current implementation to be too restrictive. Thus, we think that Debus-Booch needs to provide a number of, rather than one, methods that can be selected for controlling the transition. Examples may include: 1) The modification triggers revision history recording, 2) The modication triggers change notification mechanism, and 3) the modication triggers a change approval process. These example methods support different degrees of the control over the design process.

In the second experiment we used Debus-Booch to develop a design for the problem reporting system as described in [Boo91]. This project fits five characteristics of the Diagram Oriented Example (See Table 1). The system is to be coded in C++, has minimum document requirements, and is not safety-critical. It is an information processing system and well-understood. The design team, including the lead author and a software engineer, however, is more experienced than that described in the Diagram Oriented Example.

In this experiment our experience were similar to those in the first experiment. One additional, interesting experience is that for this well-understood domain (e.g., design of a relational database schema), the process (the Diagram Oriented Process (DOP)) could have been designed to be even more specific and therefore to provide more effective guidance. For example, Steps 1.2.2 and 2 should provide guidance to the normalization of the classes. This seems to indicate that for building a large system, an SDM might need to be tailored into a set of different processes, each of which is most effective for designing certain kinds of components of the system. For example, a large system might contain both an embedded system and a data processing system. That being the case, both DOP and TOP processes might need to be applied to developing this system.

We have installed a version of Debus-Booch at Siemens Corporate Research (SCR). Some technologists there have used the prototype and evaluated it. These technologists are specialized and experienced in evaluating CASE tools and making recommendations to Siemens operating companies. During their evaluation, the technologists executed the tool and examined all its important features. Based upon their experience, the technologists believe that Debus-Booch should be particularly useful for novice designers because the tool explicitly supports BOOD's concepts and processes. Their experience tells us that novice designers are much more interested in using a well defined, detailed process to guide their design. A tool, such as Debus-Booch, that explicitly supports an SDM process should help them to learn the SDM quickly.

Some experiences coming out of these experiments and evaluation are:

1. *Process execution hierarchy (the tree of drivers and panels in Fig. 10) cannot be too deep*: There are two main reasons for having this suggestion: 1) A deep execution hierarchy needs too much effort in tracking the detailed process states. This problem is similar to the "getting lost in hyperspace" problem found in hypertext system [Con87]. 2) Need to minimize the time overhead from transiting between various tools that support various design steps.

   These suggestions clearly reinforce our observations about the problem of mental and resource overhead [SO93]. Novice designers are more willing to accept the overhead to trade for more guidance while skilled designers are not. However, the evaluation seems to indicate that even for the novice designers, the process execution tree cannot be too deep. The evaluation suggested that three levels seem to be maximal.

2. *Designers had difficulty in selecting processes*: Users need stronger support for selecting processes. The textual help message associated with each process seems to be not sufficient. A more readable and illustrative method must be developed to help users to understand the process requirements quickly, and thereby help users to select appropriate processes.

3. *Support the coordination of designers working at different steps:* Our model focuses on supporting designers to work in parallel in designing different software components, or supporting an individual designer to work in parallel on multiple software components. However, the current model is weak in coordinating two designers working on the same software component at different process steps. For example, we found that a finished class diagram might need to be passed to another designer for defining its module diagram. This often helps in utilizing the different skills of designers.

4. *Need to have stronger support for tracking and coordinating processes*: This suggestion is closely related to the first suggestion. The evaluation indicates that the process tracking mechanism is even more important when the process guides designers at the relatively low levels of the process. The process tracking must emphasize indicating the current state of the process and help designers understand the rationales and goal for performing the step.

5 Summary

Our work in developing elaborations of the BOOD architecture into more precise design process designs and code has brought a number of technical issues into sharpened focus. Generally, we have found that it is quite feasible and rewarding to develop design processes down to the level of executable code. Doing so raises a number of key issues that are all too easily swept under the rug by process architectures and process models. Many of these issues have tended to be resolved informally and in ad hoc ways in the past. This has stood in the way of putting into widespread practice superior software design processes. The following summarizes some of the more important and interesting findings of this work.

5.1 The Advantages of Detail in Process Definition

Process modelers often struggle to choose between general process definitions and specific process definitions. Processes that are too general are often criticized for providing no useful guidelines. Processes that are too specific are often criticized as leaving no freedom to designers. We found that starting with a specific SDM such as BOOD, and then elaborating it and making it more specific to the needs of a particular situation represents a good blending of these two strategies. Doing this serves to make the resulting process sharper and more deterministic, and thus helps to make it more systematic and susceptible to computerized support. It seems worthwhile to note that taking this approach is tantamount to pursuing the process of developing a software design process as a piece of software, guided by a set of process requirements and an assumed architectural specification (in this case the BOOD architecture)

We are therefore convinced of the importance of dealing with the details when elaborating design process architectures into designs and code. Here we summarize these process design issues, and describe how we addressed them in using our approach:

1. **Step selection:** An SDM often describes many "you could do" activities in its process description. In our work we turned many of them into "you should/must do" or "you should not do" activities in order to provide more effective guidance. For example, BOOD suggests specifying module diagrams. However, when using an implementation language that directly supports programming classes and objects, Debus-Booch guides designers to not specify these diagrams because they are useless in this specific application (see Fig. 4).

   With our process programming approach to the elaboration of specific processes we also found it straightforward to specify how to incorporate various other related processes (e.g., reuse, prototyping) into the design process (see figures 3, 5 and 7 for example).

2. **Refinement selection:** An SDM generally provides its guidance as a set of high-level steps. Each high-level step has a set of guidelines. Designers are often left free to follow the guidelines

closely or rely more upon their experience. Novice would tend to follow guidelines while skilled designers would rely more on their experience with some support from the guidelines. With our approach, we provide both supports to novice and skilled designers. Novices can use the detailed process support to guide their design activities, while more skilled designers use only high-level process support.

3. **Control condition selection:** An SDM usually does not specify strictly how design changes should be managed. It usually does not specify precisely the conditions under which a step can be considered to be finished. With our approach of tailoring SDMs for specific projects, we can define the conditions quite precisely. For example, for a medical system which is often safety-critical and regulated by FDA, we decide to provide more strict control (see Fig. 2) to ensure system consistency and reliability. However, our experience in using Debus-Booch shows that such control mechanism should not be enabled until the specifications (e.g., class diagrams) are stable and have been used by other software components.

4. **Control flow selection:** An SDM usually does not specify all the possible transitions between steps, instead, it only specifies those that are likely to be done most frequently. Transitions that are the most crucial ones may also be the most difficult to explain, and thus not specified sufficiently precisely. Our approach makes it far easier to add precision to the specification of transitions. For example, Fig. 8 shows the various transitions needed for modifying classes.

5. **Concurrency specification:** As noted earlier, most SDM's are intended only to specify how to support the efforts of a single designer working on one project at a time. It is clearly unrealistic to assume that this is the mode in which most designers work, and that, therefore, support for this mode of work is sufficient. In our work we adapted an architecture [SHDH$^+$91] that is capable of supporting group development. The activities which can be performed at each step allow individual designers to work on the same design in parallel.

5.2 Related Work

We have not seen any work that is similar to our approach of developing design processes as software, then analyzing and contrasting the elaborated processes, and illustrating explicitly why currently existing SDM descriptions cannot be taken directly as a completely systematic process for specific projects. Our work is unique in that it indicates how one might use the process programming approach to modeling and coding an SDM into a family of more systematic processes used for a corresponding family of projects.

It demonstrates how SDM processes can be defined more precisely. A more precisely defined SDM process is more likely to be effectively supported and thus provides more effective guidance. This experiment encourages us to be more confident in using the project-domain-specific process programming approach to solving many problems in sharpening and supporting software processes. Some work (e.g., [BN93]) studied mechanisms for supporting generic software processes. However, without studying specific generic and instantiated processes as we did in this work, the value of these mechanisms is hard to evaluate.

This work is related to other projects aimed at developing a process-centered software environment, like those reported in [MS92, KF87, MR88, Phi89, ACM90, FO91, MGDS90, DG90]. The most significant difference between these efforts and our work is that our work, targeted at specific process requirements, provides very specific strategies for supporting specific processes that emerge from the work of other acknowledged experts (in this case, these experts are in the domain of software design). For example, we provide very specific interface architecture and tool access methods for supporting SDMs and their various users. In contrast, most work in developing process-centered environments is aimed at developing general-purpose software development environments. For instance, [MR88] supports specifying any software development rules. Marvel [KF87] is a general purpose programming environment. It does not describe specifically how to provide effective guidance for using specific development method on specific kinds of projects. Another difference is that our work focuses on evaluating varied external behaviors of the system while other work focuses on the study of implementation mechanisms and process representation formalisms (e.g., [FO91]). The study of these mechanisms and formalisms is not the focus of our paper. Comparisons of our formalisms (e.g., APPL/A) to others can be found in [SHDH$^+$91, SHO90b].

6 Status and Future Work

The current prototype version of Debus-Booch is implemented using C++, Guide (a user interface development tool), and APPL/A. It incorporates StP [AWM89] and Arcadia prototypes. The whole prototype consists of about 34 UNIX processes. Each of them supports a console, driver, panel, and other tools. It was also demonstrated at the tools fair of the Fifth International Conference on Software Development Environments [1]. At present, this prototype is being enhanced by the conversion of more of its code to APPL/A and by the incorporation of new features, new design process steps, and new design processes.

We plan to carry out the following future work:

1. Focusing on more specific project domains, to elaborate still more specific process models and support environments. This should help deepen our understanding of the project domain's influences on process requirements and SDM elaborations.

2. Collecting data about how these elaborated processes are used. Based on the analysis of these data, we would be able to adjust the processes more scientifically.

3. Developing a project-domain-specific process generator. With the specification of project properties, the corresponding process definitions and its support environment might eventually be automatically generated, at least in part.

7 Acknowledgments

We thank the members of the Arcadia software environment research consortium for their comments, particularly Stanley M. Sutton and Mark Maybee for their useful comments on the APPL/A code.

We also thank those SCR researchers, particularly Wenpao Liao, who experimented with and evaluated our prototype. We are also very grateful to Tom Murphy and Dan Paulish for supporting us to continue this work at SCR. We thank Bill Sherman and Wenpao Liao for reviewing the final version of this paper.

References

[ACM90] V. Ambriola, P. Ciamcarini, and C. Montangero. Software process enactment in Oikos. In *Proc. of the 4th ACM SIGSOFT/PLAN Software Engineering Symposium on Practical SDE*, pages 183–192, Dec. 1990.

[AWM89] P. Pircher A. Wasserman and R. Muller. An object-oriented structured design method for code generation. *ACM SIGSOFT*, 14(1):32–55, Jan. 1989.

[BN93] R. Balzer and K. Narayanaswamy. Mechanisms for generic process support. In D. Notkin, editor, *The 1st International Conference on the Foundations of Software Engineering*, pages 21–32. ACM Press, Dec. 1993.

[Boo91] G. Booch. *Object-Oriented Design with Applications*. The Benjamin/Commings Publishing Company. Inc., 1991.

[Boo92] Grady Booch. The booch method: Process and pragmatics. *Computer Language*, July 1992.

[CKO92] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Comm. of ACM*, 35(9), Sept. 1992.

[Con87] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, Sept 1987.

[DG90] W. Deiters and V. Gruhn. Managing software processes in the environment melmac. In *Proc. of the 4th ACM SIGSOFT/PLAN Software Engineering Symposium on Practical SDE*, pages 193–205, Dec. 1990.

[FDA89] Preproduction quality assurance planning: Recommendations for medical device manufacturers. Technical report, Office of Compliance and Surveillance, Food and Drug Administration, 1989.

[FO91] C. Fernstrom and L Ohlsson. Integration needs in process enacted environments. In *The Proc. of the 1st Int. Conf. on the Software Process*, pages 128–141. IEEE CS, Oct. 1991.

[HKK93] S. Honiden, N. Kotaka, and Y. Kishimoto. Formalizing specification modeling in ooa. *IEEE Software*, Jan 1993.

[HLN$^+$90] D. Harel, H. Lachover, A. Naamad, A. Pnuell, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE Transaction on SE*, 16(4):403–414, April 1990.

[Hum93] W. S. Humphrey. Using the personal software process. Private communication, 1993.

[ISP88] In C. Potts, editor, *Proc of the 4th International Workshop on the Software Process*, May 1988.

[JKLW90] Jr J. Kirby, R. C. Lai, and D. M. Weiss. A formalization of a design process. In *Proc of the 1990 Pacific Software Quality Conference*, 1990.

[KF87] G. E. Kaiser and P. H. Feiler. An architecture for intelligent assistence in software development. In *Proc. of 9th International Conference on Software Engineering*, pages 180–188, 1987.

[KH88] M. I. Kellner and G. A. Hansen. Software process modeling. Technical report, Technical Report CMU/SEI-88-TR-9, May 1988.

[MGDS90] N. H. Madhavji, V. Gruhn, W. Deiters, and W. Schafer. Prism = methodology + process-oriented environment. In *Proc. of 12th International Conference on Software Engineering*, March 1990.

[1] This paper was not published at that conference and has not been published at any other conferences or journals.

[MR88] N. Minsky and D. Rozenshtein. Software development environment for law-governed systems. In *Proc. of the ACM SIGSOFT/PLAN Software Engineering Symposium on Practical SDE*, pages 65–75, Nov. 1988.

[MS92] P. Mi and W. Scacchi. Process integration in CASE environments. *IEEE Software*, 8(2), March 1992.

[Ost87] Leon J. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, March 1987.

[Phi89] R. W. Phillips. State change architecture: A prototype for executable process models. In *Proc of the 22nd Annual Hawaii International Conference on Software Engineering, Vol II. Software Track*, pages 154–164, Jan. 1989.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[RC92] D.J. Richardson and J. Chang. Rebus requirements on elevator control system. Available upon request, Dec. 1992.

[RG92] K. S. Rubin and A. Goldberg. Object behavior analysis. *Communication of ACM*, (9):48–62, Sept. 1992.

[SHDH+91] S. M. Sutton, Ziv H, H. E. Yessayan D. Heimbigner, M. Maybee, L. J. Osterweil, and X. Song. Programming a software requirements-specification process. In *The Proc. of the 1st Int. Conf. on the Software Process*, pages 68–89. IEEE CS, Oct. 1991.

[SHO90a] S. Sutton, D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process centerd environments. In *Proc. of the 4th ACM SIGSOFT/PLAN Software Engineering Symposium on Practical SDE*, pages 206–217, Dec. 1990.

[SHO90b] S. M. Sutton, D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 206–216, Irvine, Dec. 1990.

[SM92] S. Shlaer and S. J. Mellor. Real time recursive design. Available from authors, 1992.

[SMOH91] X. Song, M. Maybee, L. J. Osterweil, and D. Heimbigner. Rebus: A requirement specification process program. Technical Report 91-17, ICS Dept. University of California, Irvine, April 1991.

[SO89] X. Song and L. J. Osterweil. Debus: a software design process program. Technical report, Arcadia-document, UCI-89-02, April 1989.

[SO92] X. Song and L. J. Osterweil. Towards objective, systematic design-method comparison. *IEEE Software*, pages 43–53, May 1992.

[SO93] X. Song and L. Osterweil. Challenges in executing design process. In *Preprints of the 8th International Software Process Workshop*, March 1993. Available from the authors upon request.

Appendix:

Challenges in Executing Design Processes[2].

Xiping Song and Leon J. Osterweil
Information and Computer Science Department
University of California at Irvine
Irvine, CA 92717
song@ics.uci.edu, ljo@ics.uci.edu

A Motivation

Executing a software process is an activity that results in the creation of software that provides guidance and assistance to humans who use the corresponding process as a guide in carrying out software activities.

Execution of software processes has been a very popular topic in the software process and software development environment communities. In the past five years, many papers [ISP88, MS92, SHDH+91] have been published to address this topic. We surveyed these papers and found that many of them focus on the study of mechanisms: process representations and architectures for representing and supporting such execution. There is little work that describes and analyzes external behaviors of an executing process and evaluates these behaviors. Previous work has demonstrated the execution of project management processes, group coordination processes, and general decision making processes. We note, however, that some key software processes, like software analysis and design, have not been executed successfully.

The software design methodology (SDM) and CASE communities have suggested that SDM support tools should support not only the drawing of SDMs' notations, but also the execution of the processes suggested by SDMs (we refer to this kind of processes as *a design process* in this paper). However, up to now, we have not actually observed a SDM support tool that supports these processes effectively.

Thus, we believe that using recently developed software process technologies to execute real design process should benefit all these communities. Executing a design process should help us to understand the issues involved in software process execution, and to evaluate design technologies from users' perspectives.

[2]**Accepted and published in Preprints of the 8th International Software Process Workshop**

B Research Goal

In this paper, we focus on a specific process domain, the domain of software design, and discuss the difficulties in executing software process in general. Our analyses are based upon our experiences in executing design processes [SO89, SMOH91, SHDH$^+$91], and in comparing and evaluating SDMs [SO92]. We also analyze other work in this area. We believe that identifying these difficulties will facilitate progress in executing software processes in general.

C Challenges in Executing Design Processes

Challenge 1: Design processes are too vague to be executed

One key problem is that the SDMs we have studied seem to define their processes in terms of artifact types (e.g., object type and class type) rather than instances of these types (e.g., objects and classes, for example, client, clerk, account, interface objects, or I/O objects). Since software designers create and manipulate these instances, These SDMs do not provide strong enough guidelines to these designers.

For example, Booch's Object Oriented Design (BOOD) [Boo91] suggests the use of the following process: 1) Identify classes and objects; 2) Identify semantics of classes and objects; 3) Identify the relationships among classes and objects; and 4) Implement classes and objects.

In practice, designers using BOOD will not identify *all* objects and classes, then define their semantics. Instead, they will first identify some major and important objects, then define their semantics. After that, they may do step 3 to identify the relationships among the classes and objects they have identified to obtain a system architecture. Then, they may do step 4 to experiment with the architecture to evaluate its feasibility, or go back to step 1 to identify other less important, or internal, objects and classes. Evaluating the BOOD process against this scenario, we found that the BOOD process is vague in the sense that it does not say, specifically for an application domain, what kinds of objects and classes would be more important and should be identified first. Unfortunately, though more recent SDMs offer more guidance, all SDMs still have this weakness to varying degrees.

Because of this, we expect that simply supporting existing design processes will not be likely to improve designer productivity significantly. Note that this is not an execution issue, per se. Instead, this is a process improvement challenge, indicating an obstacle to producing a usable and useful executable design processes.

Challenge 2: Design process execution requires the use of multiple mechanisms

Design processes actually consist of a diverse variety of activities, such as:
1) diagram drawing (e.g., drawing data flow diagrams).
2) knowledge acquisition (e.g., understanding the requirements).
3) analysis (e.g, identifying the system components).
4) selection (e.g., choosing objects).
5) definition (e.g., defining the semantics of an object).
6) development (e.g., a high-level development phase).
These activities are different in 1) time required to finish, and 2) degree of human involvement.

These differences should affect decisions about which mechanisms to use to execute these processes. We believe that the use of the following mechanisms will be affected:

1. **Process control mechanisms**: For example, the process control mechanisms classified in [Phi89] are a) user-initiated/user-guided, b) user-initiated/process-guided,
c) process-initiated/user-guided, d) process-initiated/process-guided. For a design activity that entails scant human involvement, we might choose to use mechanism c) or d). For an activity that involves humans intensively, we might choose to use mechanism a) or b).

2. **Process guidance mechanisms**: Examples of these mechanisms include those described in [MS92, SHDH$^+$91, FO91]: window, menu, checklist (task list), dialog box, icon, command button, process state graph, and task description window. It seems appropriate to use checklists to guide the execution of a set of long-executing processes (e.g., analysis, architectural design, detailed design). For guiding the execution of a set of short processes (e.g., specifying object name, author name, superclass name), it should be appropriate to use a dialog box.

3. **Process state tracking mechanisms**: Examples includes the process states described in [MS92]: none, allocated, ready, active, stopped, done, and broken. For a short process (e.g., define object name), we think these states are very much

sufficient (if they are not overcomplex). However, for a long and complex process, these states might be at a too low level, and thus, it might be useful to define and use states at higher levels of abstraction (e.g., analysis_done, architecture_design_done).

Challenge 3: Executing a design process incurs substantial overhead:

If no overhead costs were incurred, the executed design process would certainly be more useful than conventional CASE tools or SDEs. Unfortunately, we have observed that using current process technologies incurs the following kinds of overhead costs:

1. **Mental Overhead**: Showing process information to designers may cause mental overhead. 1) We expect that not all of this information will be useful to the designers *at any time*. The information (like that shown in [MS92, JKLW90]) may distract designers' attention from designing products. 2) Current technologies for executing process tend to use a set of deeply overlaid windows to support a process that is modeled as nested subprocesses [MS92]. Since a designer may execute a number of such processes in parallel, using these technologies will likely cause the "lost in space" problem that is typically found in hypertext systems [Con87].

2. **Resource Overhead**: Managing the process state and manipulating the user interface (e.g., window creation and deletion) incur performance overhead. These operations also take internal (e.g., memory) and external (e.g., window) space.

Challenge 4: Design processes are irregularly iterative

Like many design methodologists, Grady Booch stresses that his design specification process must be performed *iteratively* in refining and revising a software design. He explicitly states that this process is not a restrictively looping process, but instead, it allows designers to go back to previous steps, even when looping has not been finished. For example, using the Booch process, a designer, while at step 3, may like to go back to step 2 to change the object's semantics because he or she has obtained a new understanding of the object. Thus, the design process actually is not sequential, random, and iterative in the conventional sense, but rather irregularly iterative. This suggests the need for new and different process definition language semantics.

Three techniques have been used to support design specification process: a) a set of ordered prompts, b) a textual or graphical editor (e.g., vi), c) a textual template editor (e.g., a 4th generation tool). Technique a) will usually be not capable of supporting this irregularity effectively. Technique b) certainly allows designers to deal with this irregularity, however, at the cost of providing no process guidance. Technique c) is more useful than b) because it provides some guidance (e.g., the REBUS system [SHDH$^+$91, SMOH91] uses the template editor to guide requirement specification). Is the textual template editor sufficient for supporting the specification process? Could we develop something better than that?

Challenge 5: The need to accommodate designers of different skill levels

Some design processes are simple in the sense of that they capture only outlines of the actual design processes. Designers can learn (memorize) these processes fairly quickly through training. Thus, support for these simple processes may not help these designers significantly.

Probably, support for a complex design process (e.g., the process defined in OMT [RBP$^+$91]) would be significantly helpful, particularly to less skilled designers. However, this will make challenges 2, 3 and 4 even more challenging.

Very skilled and experienced designers would not need process support nearly as badly as less skilled designers would. Those skilled designers know, based on their experiences and skills, when to stop one step and to go forward or backward in the process.

Designers of different skill levels need different process support. This is a distinctive characteristic of process. Note that even very skilled designers use some product support tools (e.g., those that are directly involved in production, such as, compilers and diagram drawing tools). An analogy can be found in cooking. A skilled cook still uses kitchenware, but may not use a recipe (unless the process is really complex and/or rarely used). Based on this, we believe that we must provide process support that can be flexibly adapted to accommodating different designers.

D Our strategies

To meet these challenges, we are currently developing an executable version of Booch's design process.

Its execution will use our research prototypes (e.g., REBUS) and CASE tools (e.g., StP of IDE [AWM89]). We also aim to explore new mechanisms to provide process support. For example, we plan to use trigger mechanisms [SHO90a].

In the long term, we aim to develop and support more comprehensive and complex design processes. We note that design methodologists have begun to incorporate domain knowledge into SDMs, and to develop more prescriptive and comprehensive design processes [Boo92, SM92, RG92]. We believe that an executed process would be most useful when it supports a complex and comprehensive process that incorporates some domain knowledge and greater prescription.

Acknowledgements

The authors would like to thank Prof. Debra Richardson. Numerous discussions with her helped us to shape the ideas presented in this paper.