Behavior Modeling in Design System Development

Robert F. Coyne
Ulrich Flemming
Peter Piela
Robert Woodbury

Carnegie Mellon/Building Industry
Computer-Aided Design Consortium (CBCC)
Carnegie Mellon University
Pittsburgh, PA 15213 USA

We describe the development approach for a software environment to support the early phases in building design called SEED. The combination of capabilities offered by SEED to designers is novel and includes the integrated handling of solution prototypes. We give the reasons for using an object-oriented software engineering approach in the development of the system, which starts with a comprehensive behavioral model of the system from the user's perspective based on actors and use cases. We illustrate results from the first development phase and sketch the next phases. At the time of the CAAD Futures '93 conference, we will be able to report our experience in developing a first system prototype and to demonstrate the prototype.

Key words: object-oriented software engineering, integrated design systems, architectural programming, schematic layout design.

1 Introduction

This paper reports work in progress in the development of a design support system with a broad range of generative capabilities.[1] The system is intended to bring research results closer to practice. In developing the system, we use a particular software engineering approach in order to communicate our intentions to prospective users and to assure the system's functionalities through all development phases and over a distributed development environment. We believe that the process we are engaged in and our initial experiences with it will be of value and interest to others who find themselves in a similar situation.

We call the system we are developing SEED—an acronym for Software Environment to Support the Early Phases in Building Design. *SEED* is intended to support the preliminary design of buildings comprehensively. It represents the confluence of two multi-

[1]A generative capability allows a system to take an active part in the generation of a design. In the system under consideration, the designer and the system can become true partners in the synthesis process.

generational research efforts on grammar-based design systems, which includes experience with a series of software prototypes: LOOS/ABLOOS (Flemming et al., 1988; Coyne and Flemming, 1990; Coyne, 1991) and GENESIS (Heissermann, 1992; Heissermann and Woodbury, 1993); these systems have developed to the point where the underlying formalisms are mature. We have a strong interest in finding out how useful the capabilities developed through these prototypes can be for actual design practice.

In a very general sense, SEED is intended to complement designers by (1) reminding them of things they may have forgotten; and (2) suggesting to them possibilities they might not have considered. In particular, we plan to use spatial grammars and extensive evaluation tools to achieve these objectives. The intended capabilities have no direct equivalent in existing practice as we understand it. That is, we are not planning to support a mature and well-understood process, and the use of the proposed system is highly speculative. In making "our best guess" in prototyping the system's capabilities, we plan to use *behavior modeling*[2] to communicate the intended uses of the system to our sponsors, to potential users, and to system developers. We then plan to engage in an extended process of continued design, testing, and evaluation with users.

The approach taken by us may be of general interest for two reasons: (1) the novel combination of capabilities envisioned for SEED may be interesting in their own right; (2) the development of the system may be representative of the challenges faced by other developers of research software approaching tasks in non-traditional ways that are outside the experience of the envisioned user community. Due to constraints on time and resources, current system-building efforts in design research typically do not follow a structured development process and do not employ integrated methods for analysis, design, and implementation. However, this situation may be changing[3] as research centers such as the Engineering Design Research Center (EDRC) at Carnegie Mellon are moving toward developing multi-generational research systems[4] built in close cooperation with sponsors and potential users in industry. These types of efforts are becoming increasingly important as higher- and deeper-level issues, such as the evaluation and validation of new design paradigms in practice, come under investigation (Coyne and Flemming, 1990; Piela et al., 1992). In this context, *understandability, maintainabilit*y, and *extensibility* of systems become critical to the continuity and viability of the development effort. To achieve these objectives requires, in general, sophisticated mechanisms, processes, and methods for systems and software engineering - mechanisms such as abstraction, information hiding, and encapsulation; process models such as spiral models of software production; and integrated methods for analysis, design, and implementation based on a model of the real world domain.

A variety of analysis, design, and implementation technologies accommodate smaller or larger subsets of these mechanisms or methods. Object-oriented (OO) technologies are particularly attractive in this connection because among the potential benefits most often associated with them are precisely the critical requirements listed above (Graham, 1992):[5]

[2]Behavior modeling of a system results in a high-level description of the system's functionality from a user's point of view.

[3]In fall of 1992, the EDRC first introduced a required software engineering course for its M.S. and Ph.D. students in design research: 18-869 Special Topics in CAD: Software Engineering for Engineering Design.

[4]By multi-generational research systems we mean systems that outlast a given research project and possibly its personnel and are used as the basis of, extended, or built on top of for subsequent projects and systems.

- "Object-oriented programming, and inheritance in particular, makes it possible to define clearly and use modules which are functionally incomplete and then allow their extension without upsetting the operation of other modules or their clients."
- "System evolution and maintenance problem, where maintenance is understood as adaptation to a changing problem, not just correcting errors, are mitigated by the strong partitioning resulting from encapsulation and uniform object interfaces."
- "Object-oriented systems are potentially capable of capturing far more of the meaning of an application—its semantics."

While object-oriented analysis, design, and programming methods are neither necessary nor sufficient to achieve the system development objectives outlined above (Rumbaugh et al., 1991; Booch, 1991; Graham, 1992; Jacobson et al., 1992), they have evolved precisely to support those objectives more effectively if intelligently employed.[6]

In this paper, we describe our initial experiments with the OO development process and method used in the development of the first SEED prototype. These are based on a behavior modeling concept called a *use case*, which is used throughout to enforce understandability, extensibility, and maintainability in the face of evolving requirements. Section 2 describes the general objectives of the SEED project and the decomposition of the intended capabilities into the modules of a prototype system. Section 3 discusses behavior modeling of systems as an increasingly important component of many (object-oriented) software engineering methods and processes; it then relates SEED development issues to the potential benefits of behavior modeling techniques. Section 4 outlines the particular software engineering process experimentally employed by us and describes our experience in applying the first step in the process, requirements modeling. Section 5 summarizes the results of our development process so far and concludes with our plans and expectations for the continued development of SEED.

At the time of this writing, the SEED development process is less than half complete. However, the work continues, and we will be able to make a more complete report of our experience at the *CAAD Future*s *'93* conference. By then, we plan to demonstrate a prototype of the system and will have obtained feedback from prospective users. We expect that further experience will enable us to determine more clearly those parts of the system and phases of development for which the selected process appears to be most useful—that is, when is it worth the overhead/resources connected with its application.

2 Overview of SEED

This section gives a brief introduction to SEED, its overall architecture, and the first prototype.

[5]It should be noted that Graham is not one of many current authors whose books extol the promise of OO approaches while marketing their own particular OO method. He presents a balanced and thorough analysis of OO methods including their potential *disadvantages* and the conditions required to realize their benefits.

[6]OO technologies do not enforce the appropriate use of mechanisms and methods. Their effective use is dependent on many factors including experience, training, appropriate organization and management contexts, and the careful choice of projects in which they are introduced (Goldberg, 1992).

2.1 *Overall Goal*

SEED is a software environment to support the early phases in the design of buildings. It is intended to support, in particular, the design of *recurring building types*, that is, building types dealt with frequently in a firm or institution. Such organizations, from housing manufacturers to government agencies, accumulate considerable experience with recurring building types. But capturing this experience and its reuse are supported only marginally by present CAD systems. The goal of the system is to support, in principle, the preliminary design of such buildings in all aspects that can benefit from computer support. This includes using the computer not only for analysis and evaluation, but also more actively for the generation of designs, a capability that remains underutilized in present CAD systems. The intent is to develop a collection of generic capabilities that can be easily adapted to different building types.

2.2 *Approach*

Our approach for the first version of the prototype starts with the assumption that the preliminary design process can be divided into *phases*, each of which addresses a particular subtask of the overall design problem and leads to a particular set of decisions. For example, the preliminary structural design phase may deal with the task of finding an appropriate structural system for an overall building configuration. There is no assumption that phases have to occur in a strict sequence. A partial ordering between them may, however, exist because the information needs of one phase may depend on decisions made in another phase.

We plan to develop for each phase an individual *support module* based on a shared logic and architecture. This will allow us, on the one hand, to utilize various pieces of existing and possibly heterogeneous software and to distribute the development efforts among individual modules. The shared logic and architecture assure, on the other hand, that the modules appear to the user as parts of a unified whole, which includes a common style for the interfaces. They also allow us to develop generic protocols for phase transitions; they make any sequence of phases logically extensible and should facilitate the “plugging in” and “pulling out” of individual module versions. The results generated in any phase are stored in a database for reuse, either in the context of the same project or a different project.

The architecture proposed for a module assumes that the task to be accomplished in any module can be divided into five generic subtasks. Each of these is supported by a specific module *component*:

- The *input component* is the general read interface between a module and the database. This component makes, in principle, everything specified or generated by another module or by a previous invocation of the same module available to a designer. This may involve translations between data formats, or the stripping away of information. For example, if a module dealing with two-dimensional layouts reads a three-dimensional configuration of components, information about the third dimension may have to be deleted.
- The *problem specification component* allows designers to specify or modify the task to be performed in the current module. For example, if the task of the current module is to develop a schematic floor plan generated in another module into a three-dimensional configuration of building elements, this floor plan must be made available to the current module (this is the function of the input module); but the module may need additional information about the desired roof shape, wall con-

struction etc. This is done in the problem specification component.

- The *generation component* supports the generation of solutions to the problem specified for the current module. In each module, we intend to make a broad range of phase-specific support options available, from complete automation to interactive constructions that are completely under the designer's control.
- The *evaluation component* evaluates solutions for compliance with the specifications. Iterations at this stage may occur, where the evaluation results influence further generator action.
- The *output component* allows a designer to write any result generated or specified in a module into the database. Examples are a solution stored for re-use in a different project, or an intermediate solution for the current project that is to be elaborated in another module.

Figure 1 shows the data flow between these components, the user, and the database. This overall architecture reflects a bias on the part of the developers of the system. However, the bias is not that architectural design can be equated with problem-solving, not even that problem-solving is the most important aspect of architectural design. The bias is that the *problem-solving aspects are the easiest to support with computers*, at least if tools such as LOOS/ABLOOS or GENESIS are to be employed in the process. Thus, we provide a specification component that supports the explicit specification of constraints and criteria, and an evaluation component that is able to test if these constraints are satisfied or if the criteria are taken into consideration. There is no assumption that these are the only or most important design considerations. The generation component therefore allows designers to control the process of form generation to any extent desirable, and the output component allows them to save any result, no matter how many explicit constraints are violated.

In the context of SEED, these premises give us a basis from which we are able to conceive and develop a common logic and style of interface for the individual system modules that make them appear to the user as part of a unified system in the desired way. They allow us also to find an integrated way of handling the broad range of capabilities we plan to provide through the system, from the reuse of past solutions to the construction of customized solutions under the designer's control, and from problem specification to generation and evaluation through all modules. We stress, in particular, the way in which past solutions are handled: they are stored in the database as problem-solution pairs, where the problem part serves as index for retrieval when the designer faces a problem with similar characteristics. After retrieval, the associated solution is immediately available for editing and adaptation using the full capabilities of the system. Conversely, any solution generated with the system can be saved in the database with the corresponding problem statement for reuse.

2.3 First Prototype

How this works concretely is illustrated by the three modules planned for the first prototype of SEED:

1. **Architectural Programming**. A problem is posed in this module by a statement of the client's overall goals, a description of the site, etc. The module supports the generation of an architectural program for a building that realizes the client's goal; this is a solution in the context of the module. A solution that could be reused and adapted would be a program that has been developed in the past for a similar building type and site.

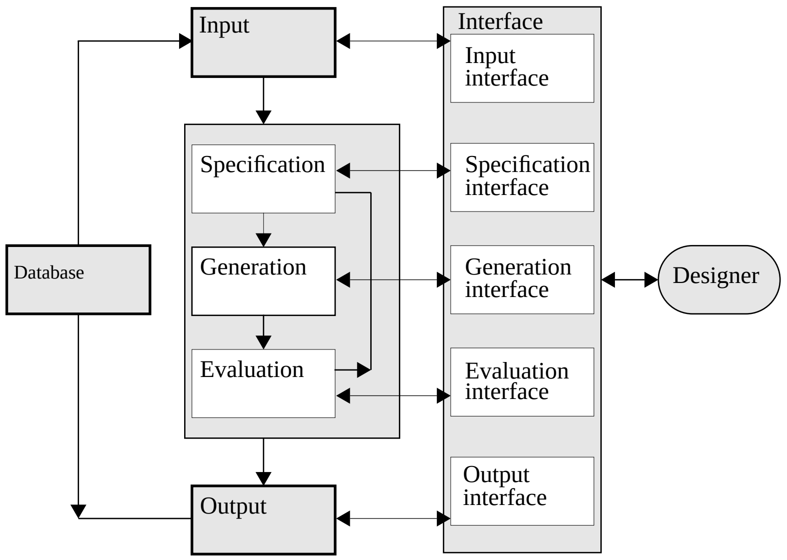

Figure 1. Data flow in generic SEED module.

2. **Schematic Layout Design**. In this module, a problem is posed by an architectural program, and the solution generated is a schematic layout of the functional components of the program. A layout that has been developed in the past for a similar architectural program can be reused. The generative capabilities in this module are based on the representation and operators developed for the LOOS/ABLOOS systems (Flemming et al., 1988; Coyne and Flemming, 1990; Coyne, 1991).
3. **Schematic Configuration Design**. The problem is posed by a schematic layout, and a solution develops this layout into a three-dimensional configuration of spaces and building components. Solutions can be used as in the other modules. The generative capabilities in this module are based on the representation and operators developed for the GENESIS system (Heissermann, 1992; Heissermann and Woodbury, 1993).

A more detailed functional specification of the first three SEED modules from a high-level user perspective can be found in Flemming and Woodbury (1992).

3 Behavior Modeling in System Design

A full discussion of the rationale and state of the art of behavior modeling as a concept and practice in software engineering is beyond the scope of this paper. Many related and overlapping concepts and techniques *have been proposed* to produce a high-level description of a system whose behavior is being modeled. In this section, we give an overview of some of these concepts and their potential in the SEED development context.

3.1 A Use Case by Any Other Name - Is What?

Use Cases. *A use case describes a specific way to use a proposed system* (Jacobson et al., 1992). A use case-driven approach to development is based on the assumption that a system can be described in a number of different views or perspectives, each of which corresponds to a set of related functional requirements. The first task in describing a system is thus to identify all of the different views of the system of interest; taken together, they comprise a complete picture of the functional requirements of the system. The use case approach differs from other view-oriented methods such as functional decomposition because it attempts to look at the system from the outside, from the point of view of its users. The first step in development is to identify the possible users of the system, and for every user, all different ways he/she must be able to use the system. The result is a set of use cases which, taken together, represent everything the users can do with the system.

The use cases guide development through all phases. They make complex systems understandable, but do not require that the system be structured into modules or objects with well-defined interfaces from the outset. Such structuring creates very technical descriptions that tend to shift the focus from system requirements to implementation-level descriptions. Use cases are identified during requirements analysis and thereafter serve many roles. They are used to structure the different models produced (analysis, design, implementation, testing, etc.) into manageable views and to relate these models to one another.

Scripts. Rubin and Goldberg describe a method for object-oriented analysis called Object Behavior Analysis (OBA) (Rubin and Goldberg, 1992). *The approach relies on scripts that record the use of the (proposed) system.* OBA is part of a larger process model incorporating the specific engineering opportunities introduced by OO technology. The motivation for OBA is that a more effective way for finding objects in OO system development is needed. The approach emphasizes first an understanding of what takes place in the system, which leads to the system *behaviors.* The approach next assigns these behaviors to parts of the system and tries to understand who initiates and who participates in these behaviors. Initiators and participants that play significant system roles are recognized as objects, and are assigned the behavioral responsibilities for these roles.

Scripts appear to share many properties with use cases and to be used in similar ways. They describe the intended system behavior from an external perspective. They help conduct the analysis of a problem situation, and they acknowledge that the outcomes of analysis must be captured in some explicit notation for purposes of communication. They also help identify the objects in the system. However, scripts have not been widely known nor have they been publicly available until very recently. Currently, specialized, integrated tools to support OBA are under development.

Scenarios. *A scenario is a sequence of events that occurs during one particular execution of a system* (Rumbaugh et al., 1991: 86). The scope of a scenario can vary; it may include all events in the system, or only events impinging on or generated by certain objects in the system. A scenario can be the historical record of executing a system or a thought-experiment of executing a proposed system. Rumbaugh et al's method (OMT) uses scenarios first in the context of constructing a dynamic model of the system (this comes after the standard object-modeling process). The following steps construct a dynamic model: (1) prepare scenarios of typical interaction sequences; (2) identify events between objects; (3) prepare an event trace for each scenario; (4) build a state diagram; (5) match events between objects to verify consistency (p. 170).

The scenarios in the OMT method have a more limited scope of application than use cases. They are used primarily for constructing the dynamic model (of object interaction) and are more technically defined in terms of an already existing object model; they are not used to help identify the initial model. It is not clear whether they will be used in the future more comprehensively through all phases of the development as are use cases.

Mechanisms. Mechanisms are *a structure whereby objects work together to provide some behavior that satisfies a requirement of the problem* (Booch, 1991:17, 149). They specify means or patterns of interactions, whereby objects collaborate to provide some higher level behavior. A mechanism represents a strategic design decision about how collections of objects cooperate. Booch's approach first identifies the key abstractions (classes and objects that form the vocabulary of the problem domain) to form a model of reality; only then does it add behavior to these abstractions to derive the observable behaviors of the system (p. 148). (How the behavior realized by a mechanism comes about is absolutely immaterial to the system user.) These strategic decisions must be made explicitly; the most elegant, lean, and fast programs embody carefully engineered mechanisms (p. 132). Identifying commonality in mechanisms can lead to simpler designs.

Mechanisms are a more technical interpretation of behavior modeling and are used in a more narrow scope of the OO development process. They are identified after the object model is created in order to relate and structure collections of objects to perform the system's required functionality. It is not clear what role, if any, they are intended to play in more upstream phases (requirements analysis) or downstream phases (testing) of the development process.

Walk-throughs. In the object-oriented design method of Wirfs-Brock et al., walk-throughs are used to flesh out and check a model of a system in terms of what happens when a user interacts with the system (Wirfs-Brock et al., 1990:32). They involve testing various scenarios to help determine what behavior needs to be distributed among classes and subsystems, and how those classes and subsystems will work together to provide that behavior. A walk-through consists of the following steps: (a) postulate a state for the system ; (b) propose an action by the user; and (3) note each class responsible for performing an action and the classes with which it interacts. It is suggested that walk-throughs can be used to help identify classes, subsystems, responsibilities, collaborations, and contracts. According to the authors, "they encourage designers to imagine how the system will be invoked, and go through a variety of scenarios using as many system capabilities as possible" (p. 63).

Walk-throughs are not used explicitly to help create an initial system model nor to identify objects, but to verify and refine these models. They are used only from the design phase on; it is assumed that the requirements for a particular program are given. However, the authors state that "a good requirements specification describes what the software can do and what it cannot do" (p. 9) and that a walk-through "is a view of the operation of the system as we have thus far conceived it. Walking through our design at this stage allows us to determine if we have left any responsibilities undiscovered or misassigned." They encourage that various (use-case like) questions are posed in terms of what happens in the system when the user does this and that, etc. (p. 80).

Other concepts. Many other concepts relate to behavior modeling such as *event partitioning* (Martin and O'Dell, 1991), and *threads* (of execution), a notation that is specialized for modeling behavior of concurrent and real time systems (Buhr and Casselman, 1992). Discussions and presentation at the OOPLSA '92 Conference suggest that there is a trend toward the more widespread use of behavior modeling through the entire OO development process—requirements, analysis, design, implementation, testing, support and maintenance; see, for example, the summary from the conference workshop "Experiences

Use Cases and Similar Concepts" [forthcoming in the Addendum to the Proceedings] (OOPSLA, 1992).

Discussion. Nearly every OO method considers behavior modeling in some form and at some stage of the development process. Most methods also agree that behavior modeling must be informed by a more or less comprehensive view of the functionality of the system from an external point of view, although at present, it is used in this way only implicitly in most software development methods. Jacobson et al. (1992), Rubin and Goldberg (1992), and, it appears, a growing number of other OO methodologists and practitioners, believe that it should be done explicitly at the very start of a software development project because in every such project, something intuitively similar to use cases must be identified if the project is to be successful. Constructs similar to use cases are identified at the latest when integration testing is performed and documentation (user's manuals) is written because these activities are by nature case-based. Identifying them early can help to guide and manage (by partitioning) the whole development work.

To date, use cases reflect the most explicit and comprehensive use of behavior modeling throughout the system development process. They are motivated ultimately by the view that the most essential property of a system is a stable structure (architecture) during its lifetime. The best way to insure this is to model the system functionality on the basis of the organization in which it will be used and its users, and to expect change,by viewing new system development as only a special case of further system evolution, development, and maintenance.

3.2 SEED Development Issues and Behavioral Modeling

> The requirements for a system will always be in a state of flux. Management or clients may impose an artificial freezing of requirements at a particular point in time. But the true requirements, the needed system, will continue to evolve. Many forces affect this ever-changing requirements set: clients, competition, regulators, approvers, and technologists. As Gerhard Fischer (1990) points out, "We have to accept changing requirements as a fact of life, and not condemn them as a product of sloppy thinking" (Coad and Yourdin, 1991).

Our emphasis on up-front analysis and design of capabilities may give the impression that we advocate "big-stick" computing—the imposition of rigidly predetermined functionality on users. The opposite is true because we are aware that this approach usually fails, especially with capabilities as novel as those we propose. Many software engineers agree that innovative development should proceed cautiously and incrementally by eliciting guidance and feedback from end-users through a series of prototypes. Our up-front analysis and design efforts are intended to make the first version of our system as sound as possible. Once a system is in the hands of users, initial suggestions tend to respect the basic assumptions of the starting point because it is hard to imagine whether a radically different system would have been a better starting point. On the other hand, many small improvements are often immediately obvious and need to be implemented in order to give the basic idea a fair trial. Thus, a new system will tend to evolve conservatively at first. If, after a while, this starts to seem futile, more radical changes will be suggested and accepted. We expect to work with the basic assumptions of the first version of our system for some time. That is one reason why we are considering those basic assumptions carefully.

Furthermore, we are proposing new computational capabilities that might be useful and plan to respond pragmatically to the suggestions and wishes of end-users. To start this dialogue, we need to describe these capabilities to practitioners, and we need to communicate to them the comprehensive scope of the system, lest they misunderstand the necessarily limited part of the system shown to them initially. They can tell us then how we might move the capabilities towards practicality. We have decided to put forward a tangible CAD tool as a starting point.

We are also optimistic that with careful consideration, we will get many of the basic assumptions correct. After all, the generative capabilities of SEED are based on the earlier prototypes of ABLOOS and GENESIS. This is not to say that we expect to have a practical system in the first version. We expect to have a basically correct, but incomplete functionality. We can predetermine some parts of a practical system without end-user participation, but not a sufficient number of parts.

To summarize, we are introducing in this project a technology which implies that practice will change. With this in mind, our process involves the following complementary steps: (i) use of explicit behavior modeling for requirements modeling/specification; (ii) controlled introduction into the work place; and (iii) evolutionary prototyping with incremental adaptation.

Initially, we expect to gain the most from using behavioral modeling in its most generic sense, for the purpose of amplifying communication among people involved in complex problem solving and decision making activities. As described in Section 2, the design capabilities embodied in SEED are decomposed into independent but cooperating modules, and there must be a clear line of communication between those modules. In general, such a loosely coupled design supports ease of change and reflects the desirable condition (and a pragmatic necessity for our effort) that the development team should also be loosely coupled (Goldberg, 1992).

4 SEED Development Process

In the development of SEED, we are experimentally following the Object-Oriented Software Engineering (OOSE) process and method developed by Jacobson et al. (1992). The present section describes some details of the approach and its relevance for the SEED effort.

4.1 Object-Oriented Software Engineering

OOSE is distinguished from other current OO approaches primarily through its reliance on behavior modeling to identify and classify the objects in the system. Most object-oriented analysis and design techniques claim that the best and most stable systems are built by using objects that correspond to real-life entities. OOSE augments this practice by an object model based on three object types: entity, interface, and control objects. The expectation is that the three object types provide greater stability for the model because changes will be localized. For example, behavior that is placed in control objects will in other methods be distributed over several domain objects, which makes it harder to change this behavior.

In OOSE, the behavioral description is based on *actors* and *use cases.* Actors model prospective users, everything external that is to communicate with the system, including other systems. Actors represent a certain role, or class description of behavior, rather than an actual person who uses the system. Actors are the main tool for finding **use cases**, and together actors and use cases define the complete functionality of the system.

Each use case is a complete course of events in the system from the user's perspective. The use cases are the central thread running through the whole of OOSE.

OOSE views system development as *model building*. It includes three main development processes: analysis, construction, and testing. Each of these processes produces one or more associated models: the analysis process produces the requirements model and the analysis model:[7] the construction process produces the design and the implementation model; and the testing process produces the testing model.

Thus, OOSE specifies a series of models as products (milestones) in the system building process. The production of each of these models constitutes a phase within the overall iterative process:

- **Requirements Model.** The requirements model aims at delimiting the system and defining what functionality the system should offer. Actors and use cases are defined in this model. It serves as a means of communication between the developers and orderer of the system. It thus describes the developers' view of what the customer wants and should be readable for non-OOSE practictioners.
- **Analysis Model**. When the initial requirements model has stabilized and has been approved by the users or orderers of the system, the actual system development starts by developing the analysis model. This model aims at structuring the system independently of the actual implementation environment. In this model, the aim is to capture information, behavior and presentation in these respective object types: *entity (i.e. domain) objects, interface objects,* and *control objects*. The analysis model derives from the requirements model and forms the basis of the system architecture.
- **Design Model**. The design model refines and formalizes the analysis model. It is the first model that takes into account the actual implementation environment of the system. It defines explicitly the interfaces of objects and the semantics of their operations. Other important system environment issues are also handled such as DBMSs, programming language features, distribution, and so on. The first attempt at a design model can be made mechanically based on the analysis model in order to ensure a clear traceability in the models.
- **Implementation Model.** The implementation model consists of the actual source code composed or written for the system. It implements each specific object specified in the above models.
- **Test Model**. The test model is developed to support the verification of the system developed. This involves mainly documentation of test specifications and test results. Testing starts with the lower levels (unit testing), in order later to cover the use cases and finally the whole system (integration testing). Thus, the requirements model supports and is verified by the testing process.

In the actual development process, these models will undergo many changes before they become stable. Generally, work on a successive modeling phase is not initiated until results from the previous phase have stabilized. However, the sketching of models in subsequent phases and the revisiting and refinement of the models of previous phases will

[7]This model appears to have a poorly chosen name because it is only part of the overall analysis process, which also includes the requirements modeling phase. The analysis model is in fact a more robust and refined version of the requirements model defined in terms of initial descriptions of domain, interface, and control objects. A better name, perhaps, would be the analysis-object model.

sometimes be indicated by the work on a current phase. Each of these modeling phases has a method associated with it, which is comprised of certain steps to be taken in building and refining the model. The steps have a certain ordering based on experience and pragmatic concerns. But the process within each phase is not strictly linear, and the method for eachmodeling phase is also applied in an iterative fashion.

To the extent that time and resources allow, the SEED project will follow the OOSE process through its various modeling phases and methods. We believe that the most important part of the process is the initial high-level behavior modeling that establishes the requirements of the system. This is true, perhaps especially so, for the development of research-based design systems. As articulated in Section 3.2, we are exploring the hypothesis that for such systems, their functionality and the expectations and interactions of users can and must be carefully hypothesized, conjectured and prototyped, and that an object-oriented, use case-driven approach effectively supports that process.

System Level
Operate SEED

Input Component Level
Read Architectural Program from Database
Read Layout from Database
Read Problem Statement-Layout Pair
  from Database
Retrieve Case from Database

Problem Specification Component Level
Select Active Problem Statement
Add Functional Unit
Delete Functional Unit
Edit Attributes of Functional Unit
Aggregate Functional Units in Problem Statement
Disaggregate Functional Units in
  Problem Statement
Copy Problem Statement
Delete Problem Statement

Generation Component Level
Select Next Generation Event
Select Active State
Add Design Unit Under Designer Control
Change Spatial Relations Between Design Units
Change Function of Design Unit
Edit Dimensional Attributes of Design Unit
Remove Design Unit
Edit Target
Edit Execution Parameters
Generate to Target
Copy State
Delete State

Evaluation Component Level
Evaluate Layout
Edit Evaluation Parameters

Output ComponentLevel
Save Problem Statement
Save Layout
Save Problem Statement-Layout Pair
Save Case

Figure 3. Current Catalogue of Use Cases for SEED-LOOS.

4.2 Use Cases in the Development of SEED

An integral part of the development of use cases is a glossary of key terms and concepts, which forms the basis for communication between developers, sponsors, and potential users, both within and across any of these groups. Later, the glossary of concepts becomes a rich basis for the identification of objects in the system. In the present section, we present selections from the glossary developed for SEED and examples of use cases that represent a slice of the anticipated functionality of the system, taken from the requirements model developed for one of its modules, SEED-LOOS, which supports the schematic layout design phase. These selections include necessarily concepts that have significance for the overall system and other modules. Figure 3 lists the use cases that have been modeled thus far for SEED-LOOS

4.2.1 Selected Entries from the Glossary of Key Terms and Concepts

This section presents entries from the current glossary that are needed for an understanding of the following use case descriptions.

Client Program. A client program specifies the client's goal in terms of what is to be built where, when, at which cost, and under which expectations. An explicit program should state at least the following:

- site and site-related restrictions, specifications, etc.;
- building type and overall indication of size (e.g., total square footage of rentable space for an office building, total number of students for a school, or number of bedrooms for a residence);
- budget; and
- references to applicable codes, standards, and regulations.

Architectural Program. An architectural program elaborates the client program in terms of the following major parts:

- specification of context (e.g. site characteristics);
- specification of main functional units needed to achieve the client's goal; and
- references to the applicable building codes, standards, and regulations.

Functional Unit. A functional unit is an identifiable object intended to perform a specific function or combination of functions in a building (e.g., a living room, a wall). A functional unit has associated constraints and criteria on its shape, size, placement, relations with other functional units etc.

Design Unit. A design unit is a part of the spatial or physical structure of a building with an identifiable spatial boundary. In a formally complete schematic layout or three-dimensional configuration, each design unit has a functional unit associated with it.

Phase. A phase is a subprocess in the overall design process that addresses a specific task. It is characterized by the type of problem it addresses and the type of solution it produces; e.g., the schematic layout phase transforms an architectural program (which describes the problem to be solved) into a schematic layout (the solution to the problem). Phases are partially ordered based on the information they generate.

System Module. A module of the SEED system provides software support for an entire phase.

Module Session. A module session is a specific invocation of a SEED module. It has a start and an end, and operates on run-time memory that is separate from the database. Data in run-time memory are called *available*, and available data on which the designer can operate are called *active*.

Database. The database is a collection of information that supports modules and their components. Unless otherwise noted, the term *database* refers to two types of information:

- project-independent information stored for reuse across projects (e.g., cases); or

- project-specific information for reuse across sessions (e.g., design versions, design histories).

Problem. In the context of a module session, the term *problem* refers to the problem to be solved in the session.

Problem Statement. A problem statement is the representation of a problem in a module session.

Design Space. In general, a design space is the set of all (partial or complete) solutions to a design problem together with some structure that allows us to navigate through the set. In the context of the present module (as well as other modules such as SEED-GENESIS), a design space is *implicitly* defined by the combination of a problem statement, a starting state, and a collection of operators that derive states from states, where the states represent partial or complete solutions of the problem given in the problem statement. The space is the collection of all states that can be derived by applying the operators to the starting state under the problem statement. A (part of a) design space is *explicitly* defined by a collection of explicit states.

State. A state in a design space is a starting state or the result of applying operators to a starting state, where each state represents a partial or complete solution to the given problem.

Solution. In the context of a module session, the term *solution* refers to a solution to a problem to be solved in the session. The solution of one phase may become (part of) the problem statement of another phase.

Partial Solution. A partial solution is an incomplete solution in terms of the problem statement of a session.

Problem-Solution Pair. A problem-solution pair is a problem statement and one related solution.

Problem-Solution Tuple. A problem-solution tuple is an ordered set, or “chain,” of problem-solution pairs, where the solution of one pair defines (part of) the problem of the next pair.

Case. A case is a problem-solution tuple that is stored in the database for reuse. It is minimally indexed by the problem part of the first pair of the tuple and may contain natural language annotation. In phase 1, a client program (problem) stored with an architectural program (solution) may be a case. In phase 2, an architectural program (problem) stored with a schematic layout (solution) may be a case. A related client program, architectural program, and schematic layout may be a case that extends across phases.

Layout. A layout is a collection of non-overlapping rectangles with sides parallel to the axes of an orthogonal system of Cartesian coordinates (see Figure 2), where each rectangle represents a design unit. SEED-LOOS is restricted to this type of layout. In SEED-LOOS, the internal representation of a layout of rectangles comprises the following parts:

- a set of realizable above/below and left/right relations between the given rectangles so that exactly one relation is defined for any pair of rectangles in the layout (see Flemming eta al., 1988 for details); *realizable* means that there exists at least one layout of rectangles with exactly these spatial relations;
- upper and lower bounds on the dimensions of each rectangle as implied by its spatial relations with the other rectangles in the layout; and
- the functional units associated with each rectangle.

Note that this representation does not depend on an underlying grid and does not depend on exact coordinates. It is thus able to support the rough and tentative dimensioning characteristic of the early layout stages in design.

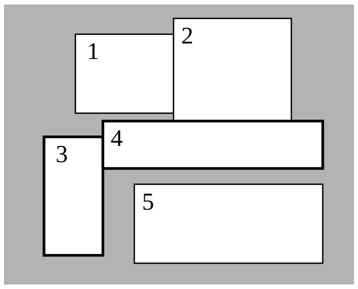

Figure 2. An example layout.Figure 2. A layout.

Score. A score records the result of evaluating a layout.

Target. A target is a predicate consisting of logically connected conditions that stops the automatic generation of layouts when the conditions are satisfied. The conditions that can be used are the following:

- a list of the functional units to be allocated and the order of allocation (this implicitly bounds the depth of generation);
- elapsed clock time;
- score (the result of an evaluation);
- maximum number of states generated; and/or
- maximum number of alternatives that meet the other conditions.

The system supports the flexible combination of these conditions in any logical statement consisting of conjuncts and disjuncts. The target is intended to support operations such as the following:

- generate next layout; this would be specified by a target with only one functional unit in its list and a maximum number of states equal to one;
- generate all layouts of a given set of functional units with score better/worse than specified score;
- generate as many complete layouts of a given set of functional units as is possible within the specified clock time and maximum number of states; and
- generate a specified number of alternative layouts of a given set of functional units.

Execution Parameters. The execution parameters define the form of user interaction with the system during various operations. For example:

- the form and frequency of the results display;
- the destination of display; i.e., what screens are to be used for display; or
- the type of user interaction desired by the designer, for example, the system may stop after each state and wait for the designer's response before generating the next state, or proceed immediately.

Evaluation Parameters. The evaluation parameters define how and when evaluations are performed during generation. For example, if evaluations are to be performed for each partial or only for complete solutions, or if only basic constraints are to be considered or computationally more expensive simulations are to be performed. The results of an evaluation are recorded in the score of the layout.

4.2.2 Example Use Cases

The following use cases employ these concepts. They are restricted to the actions taken by the designer as opposed to other actors, e.g., the experimenter or system configurer. At this time, we have found no meaningful way of dividing the designer's role into different parts. The examples are all taken from the generation component of module 2 and give a good overview of the range of generative capabilities envisioned for all modules of SEED. The following descriptions are much shorter than those in the requirements analysis document.

Select Next Generation Event. Generation takes place through a sequence of events selected by the designer. The system's participation in the generative process varies widely with the event selected. The designer selects the event from an operations menu and requests its execution from the system.

Select Active State. The designer considers the states that are currently available and selects a state. The system supports navigation through the states by displaying a generation tree showing all states that are currently available and the parent/child relations between them. It also provides some standard operations to move from state to state, for example, "move to parent of active state."

Add Design Unit Under Designer Control. The designer selects a functional unit in the active problem statement that has not yet been allocated and indicates where the unit is to be added to the active state. The system provides several mechanisms for doing this: dragging, or picking a "wall" and units to be pushed away from the wall. The system adds a design unit in the indicated location and assigns to it the selected functional unit. It updates the positions of all other design units in the state in response to this operation; that is, the units are moved out of the way automatically. This feature is an important difference between SEED-LOOS and commercial CAD systems. The newly generated state becomes the active state.The system displays the active state.

Change Spatial Relations Between Design Units. The designer selects two units that face each other across a wall in the active state in order to change the relation to one orthogonal to it; that is, an above/below relation is changed into a left/right relation and vice-versa. Restrictions apply, and ambiguities have to be resolved.

For example, changing the spatial relation between the two units may change, by transitivity, the spatial relations between other units in the state; the dimensional attributes of all of these units, or the concurrents associated with them, must permit these changes. The system evaluates these possible conflicts. It makes the changes in the structure if they are consistent with the configuration and active problem statement. The designer may be given the option of forcing the changes if conflicts exist. The changed layout becomes the active state. The system evaluates the active state according to the current evaluation parameters.

Change Function of Design Unit. The designer selects a design unit in the active state and a functional unit in the active problem statement. The system replaces the function associated with the design unit with the selected function. The newly generated state is evaluated as indicated by the current evaluation parameters.The newly generated state becomes the active state. The system displays the active state.

Edit Dimensional Attributes of Design Unit. The system provides several means to select an attribute and to change it:

- The system displays directly the numerical attribute values, e.g., an *x*-coordinate, so that the designer can overwrite them.
- The designer may pick a side and drag it to a desired coordinate, thus changing the length of the adjacent sides along with the area and the aspect ratio of the unit.
- The designer may pick the center point and drag the entire unit to a different location without changing its dimensions.

In each case, the desired change must be compatible with the structure underlying the layout; for example, a unit that is above another unit cannot be dragged to a position where it would be below that unit. However, it can be removed and then be reinserted below the other unit. The system assures structural consistency by preventing updates that are inconsistent with this structure. These changes do not have to be permanent. For example, a designer may wish to extend the side of unit *a* temporarily to align with another unit *b*, but be perfectly willing to abandon this alignment when another unit is added that requires a contraction of *a*. We plan to handle this by a “freeze/unfreeze” mechanism that is under the designer’s control. Note that all of this leaves the constraints associated with the *functional* unit allocated by *a* unchanged. The changed configuration becomes the active state. The system evaluates the active state according to the current evaluation parameters.

Remove Design Unit. The designer selects a unit in the active state for removal. The system removes the unit. The newly generated state becomes the active state. The system displays the active state.

Edit Target. The designer requests display of the current target, selects one or several components in the target, and changes their value.

Edit Execution Parameters. The designer requests display of the current execution parameters and changes their value.

Generate to Target. The designer initiates the automatic addition of units from the active state until the current target is satisfied. Attributes of the environment including specified execution parameters control the designer's view and interaction with the generation process and results (both intermediate and final).

Delete State. The designer selects a state to be deleted. The system deletes the selected state and all states derived from it.

Copy State. The designer selects a state, and the system makes a copy of the state. The copy becomes the active state. This may be useful when the designer wants to generate an alternative that can be easily derived from the copy (in the designer's judgment).

5 Conclusion: Plans and Expectations

As we stated in the introduction, by the time of the conference we will have accumulated more experience with this approach and will be able to demonstrate a first system prototype. The following statements are therefore tentative.

Our experience so far supports the general value of the selected approach in the development of design systems (even or especially in research contexts involving multi-generational research software systems). The documentation of objects alone does not yield an understanding of a system. We have found that explicit behavior modeling constructs such as use cases promote an understanding among the system developers about the behavior of the proposed system. The modeling process draws out early on multiple views of the system, which enable us to build flexibility into the system architecture for evolution and avoids the worst drawbacks of tight coupling among components due to data and control dependencies. The approach leads to communication and negotiation with respect to terminology, functionality, etc. among developers with different backgrounds and responsibilities. These activities must and do occur in any kind of development. However, the value of an explicit behavior modeling approach is in delineating a structured iterative process for these activities and in providing a shared external memory of the process.

The high-level behavior specification provides a basis for a dialogue with users; in our case, the initial use cases are not developed with end-users, but will serve as a means of communication with sponsors (who have specific end-users in mind).

We also believe that other developers may gain insights from or be able to reuse portions of the body of use cases produced for our system. Many issues cut across different generative CAD systems, and other system designers may take our model as a start. For example, the use cases developed for layout might apply to any system involving interactive geometric construction. Their decomposition may also be useful for developing systems with broad functionalities that may be used in tandem or independently.

At the time of this writing, our next step is to engage in an iterative process of review and revision of the use case model that we have produced with our sponsors. As time and resources allow, we plan to continue with the OOSE through the subsequent phases of forming the analysis object model, design, implementation, and testing. In any case, we will continue to apply the process at least on a substantial vertical slice of the overall SEED system. For that portion of SEED, work on the analysis object model, the identification of classes and objects, has begun.

References

Booch, G., 1991. *Object-Oriented Design with Applications.* New York: The Benjamin Cummins Publishing Company, Inc.

Buhr, J.A., and Casselman, R.S., 1992. “Architectures with Pictures”, in A. Paepcke (ed.), *Proceedings,* OOPSLA ‘92 Conference on Object-Oriented Programming Systems, Languages, and Applications, October 18-22, 1992, Vancouver, British Columbia, Canada. New York: The Association for Computing Machinery, pp. 466-483.

Coad, P., and Yourdon, E., 1991. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall.

Coyne, R.F., 1991. *ABLOOS: An Evolving Hierarchical Design Framework*. Ph.D. Dissertation, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Coyne, R.F., and Flemming, U., 1990. “Planning in Design Synthesis - Abstraction-based LOOS,” in J.S. Gero (ed.), *Artificial Intelligence in Engineering. Vol. 1 - Design (Proceedings,* Fifth International Conference, Boston, MA). New York: Springer-Verlag, pp. 91-111.

Fischer, G., 1990. “Human-Computer Interaction in Software: Lessons Learned, Challenges Ahead,” *IEEE Software,* January.

Flemming, U., Coyne, R., Glavin, T., and Rychener, M., 1988. “A Generative Expert System for the Design of Building Layouts - Version 2,” in J.S. Gero (ed.), *Artificial Intelligence in Engineering: Design, (Proceedings,* Third International Conference, Palo Alto, CA). New York: Elsevier, pp. 445-464.

Flemming, U., Coyne, R. F., Glavin, T., Hsi, H., and Rychener, M. D., 1989. *A Generative Expert System for the Design of Building Layouts (Final Report),* Report EDRC48-15-89, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.

Flemming, U., and Woodbury, R.W., 1992. “High-Level Specification of a Software Environment to Support the Early Phases in Building Design,” Report EDRC48-31-92, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.

Goldberg, A., 1992. “Wishful Thinking,” *Object Magazine* 2(4) (Nov-Dec 1992), pp. 102-104.

Graham, I., 1992. *Object Oriented Methods*. New York: Addison-Wesley.

Heisserman, J., 1992. *Generative Geometric Design and Boundary Solid Grammars*. Ph.D. Dissertation, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Heisserman, J., and Woodbury, R.W., 1993. “Generating Languages of Solids Models,” submitted to *Solids Modeling.*

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach.* New York: Addison-Wesley.

Martin, J., and O’Dell, J., 1991. *Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice Hall.

Piela, P., Katzenberg, B., and McKelvey, R., 1992. “Integrating the User into Research on Engineering Design Systems,” *Research in Engineering Design*, Vol. 3, pp. 211-221.

OOPSLA ‘92, 1992. *Proceedings*, Conference on Object-Oriented Programming Systems, Languages, and Applications, October 18-22, 1992, Vancouver, British Columbia, Canada. New York: Association for Computing Machinery.

Rubin, K.S., and Goldberg, A., 1992. “Object Behavior Analysis,” *Communications of the ACM* 35(9) (September 1992), pp. 48-62.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., 1991. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L., 1990. *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice Hall.