Building and Databases: the SEED Experience

Ulrich Flemming and James Snyder
Department of Architecture and Engineering Design Research Center (EDRC)
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Introduction

The Software Environment to Support the Early Phases in Building Design (SEED) aims at providing computational support for the early phases in building design. The goal is to provide support, in principle, for the preliminary design of buildings in all aspects that can gain from computer support. This includes using the computer not only for visualization, analysis and evaluation, but also more actively for the *generation* of designs, or more accurately, for *the rapid generation of computable design representations* describing conceptual design alternatives and variants of such alternatives at an appropriate level of abstraction, but with sufficient detail that enables sophisticated evaluation tools to receive all of the needed input data from the representation.

The sponsors of SEED realize that the creation of such representations constitutes a major bottleneck in current CAD systems. This software is therefore unable to support early design *exploration*, that is, the fast generation of alternative design concepts and their rapid evaluation against a broad spectrum of relevant - and possibly conflicting - criteria, where the criteria themselves may evolve dynamically through this process (see [Flemming 1994] and [Flemming and Woodbury 1995] for an overview of SEED).

The tasks supported by SEED at the present time are architectural programming, schematic layout design and the generation of a fully 3-dimensional configuration of physical building components like structure and enclosure; each of these tasks is supported by an individual module using internally a module-specific design representation appropriate for the operations performed by the module. Common to all modules is an explicit representation of important design requirements (called a *problem specification*) that the design under development must satisfy. It is precisely the availability of such specifications that allows the modules of SEED to automate various tasks in the generation of design representation that have to be handled manually in present-day CAD systems. An example is the automated relocation and resizing of rooms in a layout when a new room is being inserted or when an existing room is being removed or changes its function.

A *database* allows designers to store and retrieve different design versions, alternatives and past designs that can be reused and adapted in different contexts (*case-based design* in the terminology of Artificial Intelligence). In addition, the database stores recurring problem specifications and typical requirements for building types or functional areas common to many buildings. It may thus function as an important mechanism to preserve the experience a design firm gains with specific building types and design problems independently of the designers who generated this knowledge initially. The database serves also as a main means of information exchange between modules, which do not communicate design decisions directly to each other.

The database thus has to provide a wide range of functions and capabilities that - we believe - are of interest beyond the SEED context. In fact, our work confronts or uncovers general issues likely to arise in connection with multi-functional, multi-user and distributed design environments that require broad and massive support from a database. We present these issues in the present paper and outline the approach taken by the SEED developers to deal with them.

The next section describes the requirements the SEED database has to satisfy in greater detail. This is followed by a section that puts these issues in the broader context of the software engineering challenge confronting the SEED developers. A review of the available tools suggests that they may address individual issues, but fail to provide an integrated, coherent framework that can be maintained, yet remains extensible. We describe the modeling language SPROUT which we use in SEED as the corner stone of a modeling environment in which the database and communication needs of SEED can be satisfied.

SEED-Database Requirements

Persistent storage of design data of interest

The primary design data of interest in SEED are the *problem specifications and solutions* that the users of any module generate or modify during a session of that module and deem important enough to justify persistent storage in the database. This includes different *versions* of these pieces of information as well as *alternative formulations* that may be used again for the same design project. Data of interest furthermore extend to information that can be re-used across design projects, like standard problem formulations for recurring design problems and standard or 'generic' solutions to these problems that can be adapted to different design contexts (if we use corresponding terms from Artificial Intelligence, the SEED database must support *case-based design* [Flemming 1994]).

It is important to note that the SEED modules generally capture and represent design information by means of object-oriented representations as they have become standard in advanced design support systems. Furthermore, this information is normally not captured in terms of isolated objects, but in terms of (possibly complex) object *configurations,* that is, networks of linked or associated objects. As an example, figure 1 depicts the objects and relations used by SEED-Layout (SL), a module in SEED that support the generation of schematic layouts, to represent a layout that is an intermediate or complete solution to an associated layout problem given in a layout problem specification (see [Flemming and Chien 1995] for an introduction to SEED-Layout). A *layout* is a collection of *design units* representing individual rooms, spaces or zones in a layout. Each design unit is linked to a *geometry object* that describes its location and shape in the layout (the present version of SL links design units only to rectangles whose corner coordinates are given as coordinate intervals to capture the looseness of early schematic layouts). A design unit also has explicit links to the design units that are immediately above, below, to the right and to the left (which facilitates spatial reasoning about 'topological' aspects of the layout). Each design unit is furthermore associated with a *functional unit* that collects the requirements which the design unit must satisfy (like width and area requirements as well as required relations to other design units). Note that a layout may be contained within a design unit in a layout at a higher level of abstraction; that is, it can be a sublayout of that layout. In this case, the sublayout maintains a link to the containing design unit, which determines its overall boundary (we will return repeatedly to this example in subsequent sections).

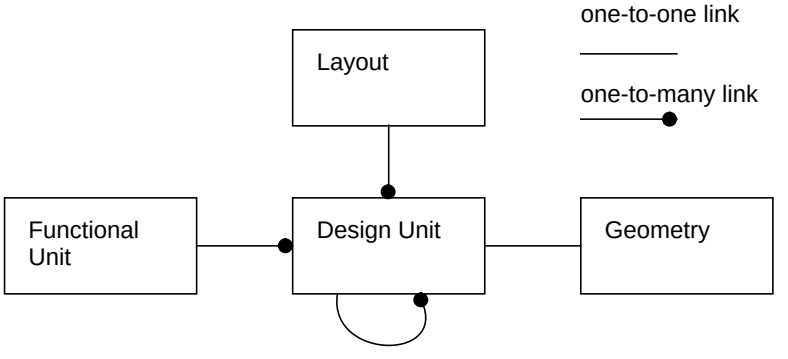


Figure 1 : Layout Represenation in SL (simplified)

Interactive browsing and retrieval of stored objects

This is the only way for modules to exchange design information. They therefore need an interface to the database that allows for browsing, where the interface appears to the user of a module as a genuine part of that module; that is, we need run-time interaction with the database through the module interfaces. This is generally referred to as *workspace management* in database terminology.

Since modules are allowed to use internally any representation that appears promising, given their task, and since we can assume that this representation differs from the database sche-

ma (which has to accommodate the information needs of *all* modules), the retrieval of objects from the database requires translation, if not schema mapping. The same is true for the reverse event, saving an object in the database.

Configuration management

Relations between objects in a configuration behave differently under different database events and must be managed appropriately. The database events that the SEED database must be able to handle properly in terms of configuration management are the following:

- *Create*. When an object in a configuration is created, the database must know which of the associated objects should also be automatically created. For example, a design unit in SL is always related to a geometry object of known type, and this geometry object should be created together with the design unit and as an integral part of it. On the other hand, the functional unit associated with the same design unit cannot be automatically created because it may already exist; it is in any case not uniquely determined and must be explicitly selected from among the available functional units by a SL operation (in interaction with the user). If linked objects are to be created, the cardinality of the relationship determines the number of objects that must be created. Additionally, prototypical objects (see below) can be used to create default objects; otherwise, objects are initialized to the defaults of their respective attributes.
- *Delete*. Conversely, when an object in a configuration is deleted, the database must know which of the associated objects should also be automatically deleted. In the above example, the geometry object associated with a design unit should be deleted with the design unit, but not the associated functional unit because it may be associated with design units in other layout versions or alternatives, or simply because it is stored persistently in the database in its own right.
- *Copy*. Copying a configuration in SEED means the construction of a complete, 'deep' copy with a new identifier. This means usually that all objects in the configuration are copied and the copies linked in a network that is isomorphic to the network linking the originals. But exceptions may exist, and it must be possible to specify these exceptions.
- *Anchor*. Anchoring a configuration in SEED means creating a different *version* of a configuration under the same identifier that can be retrieved based on a current time stamp. In this case, configuration management depends entirely on how a module intends to handle this case. When SL, for example, creates a new version of a layout, it copies its design units and makes modifications to the copies (which include addition or deletion of design units), but maintains associations to the same functional units; that is, the layout represents a different way of allocating these functional units. When the user anchors this layout in the database, she expects the database to create new design units with relations as established in the new layout, but to associate these units with existing functional units in the current problem specification.
- *Overwrite*. Overwrite replaces a configuration in the database with a configuration that has the same identifier, but a new time stamp. Configuration management proceeds similar to anchor.

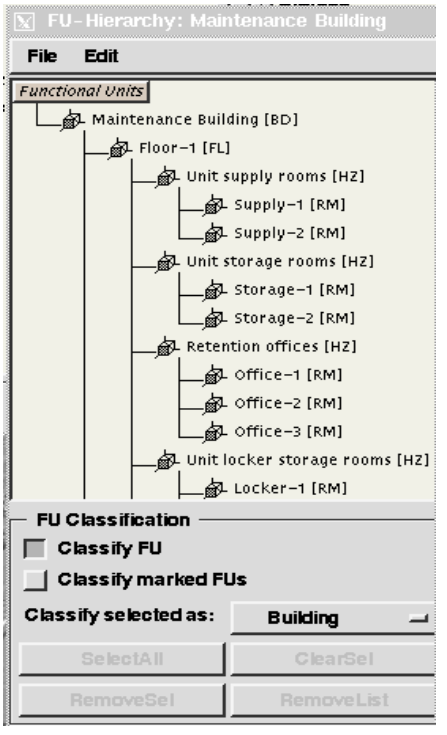
Inheritance

The objects in object-based representations typically belong to classes or subclasses, where subclasses inherit properties (attributes and behavior) from superclasses. The database supporting SEED must support this type of inheritance; that is, when an object is retrieved, it must inherit all properties and behaviors of the superclasses to which it belongs.

However, the SEED developers decided early on that the database would have to support only single inheritance because the anomalies and ambiguities inherent in multiple inheritance cannot be resolved consistently across different programming languages and object-based representations. Specifically, single inheritance is to be used carefully in a module to assure polymorphism, which is one of the cornerstones of object-based development and gives it its unparalleled modularity [Meyer 1988]. Delegation and composition techniques can be used to

complement the inherited behavior and usually realize the desired behavior overall in a more robust manner [Gamma *et al.* 1995].

We show in Figure 2 a part of the programming module interface that supports the construction of spatial configurations of functional units, where the units at higher levels contain the units at lower levels (a special form of a *part_of* relation). The interface indicates at the same time the classes to which the individual functional units belong.



Functional Unit Classes

BD - Building
FL - Floor
HZ - Horizontal Zone
RM - Room

Figure 2 : Functional Unit Classification in SEED Pro

Multiple classifications

Inheritance should not be used simply for classification purposes. However, it must be possible to classify the objects in the database through multiple, essentially orthogonal classification hierarchies. For example, a spatial functional unit as used in SL may be simultaneously classified as

- 'room' with an associated behavior during space allocation: the room 'knows' that it cannot be further subdivided by other spaces and would reject such attempts, a opposed to a 'horizontal zone' that can contain sublayouts of rooms or other zones.
- 'office' so that an energy or structural analysis program can make some plausible thermal or gravity load assumptions; in addition, 'office' may be a subclass of 'commercial' usage, which may have implications for a building code checker (see [Garrett *et al.* 1995] for a description of how standards and code checks are handled in SEED).
- 'public' or 'private' area so that a pathfinding algorithm trying to find an emergency route in a layout knows if it can move through the office or must move around it.

These classifications are essentially orthogonal. For example, offices can be public (like secretarial pools) or private (like a director's office); conversely, neither public nor private rooms have to be offices. We can thus expect that objects may belong to an arbitrary number of independent classification hierarchies. These classifications must be preserved when an object is retrieved from the database; that is, an application must be able issue *is_a* queries to a retrieved object. Conversely, the classifications must be available for database queries. For example, a private office must be retrieved by both a query for offices and a query for private spaces.

Aside from this general function, multiple classifications must support two specific features of particular importance for SEED, which are described below

Case indexing and retrieval

Reusable problem specifications and solutions should be retrievable from the database based on some form of search index; that is, the user should not be required to remember the appropriate object (for example, by name) because the object may, in fact, have been created by a different user. Again, this is a capability not supported by current CAD systems. The indices themselves may be complex object specifications, for example, a hierarchy of functional units and a design context that is used to search for layouts allocating these functional units in a similar hierarchy and a similar context. That is, indices consisting of keywords or attribute/value pairs, which are often used in case-based design, will not do for SEED.

Multiple classifications play a particularly important role in case retrieval. When a module user asks the database to find promising solutions for a current design problem, the respective problem specification should be used as index to find solutions associated with similar problem specifications. The problem specifications, in turn, may be complex object configurations similar to the example shown in Figure 2. In order to compare these types of specifications, some form of structure- or graph-matching algorithm has to be used, which is computationally expensive and cannot be done for all objects in a (presumably very large) database. We therefore plan to use classification as a filter to restrict the number of configurations that have to be structurally matched to the most promising ones, based on the class membership of the top-most elements.

Prototypes

Aside from support for case-based design, multiple classification must be available to retrieve object prototypes with standard or default properties. For example, the database may contain a room classified as ‘large office’ with standard space requirements that can be retrieved and instantiated for inclusion in a problem specification under development under the name ‘director’s office’.

Constraint management

The database should accept and manage restrictions on attribute values and relations to guarantee basic well-formedness of the *data*; for example, when a module retrieves the altitude of a sun ray at a specific day and latitude, it should not have to check if the angle is in the expected range (between 0^o and 90^o). On the other hand, we do not require the database to enforce well-formedness of the *design* in physical terms. There are significant theoretical and technical reasons for this, which we cannot explore here. Suffice it to say that the exploratory nature of design, which SEED tries to encourage, leads to design in an ‘open world’: designers may pursue ideas for a while and then abandon them in an unfinished state; they must be able to save these unfinished and possible inconsistent design states, for while at least, in the database because they may not know which of them may lead later to the final design.

SEED - Software Engineering Challenge

The requirements outlined above cannot be viewed in isolation under a narrow database perspective. They are an integral part of the overall software engineering challenge confronting the SEED developers (and anyone aiming for a similarly comprehensive design environment).

Software systems integration

SEED must integrate *heterogeneous software* developed in-house (like the modules or standards processor) or externally (like an energy simulation or cost analysis program or general-purpose geometric modeler). At the same time, it must maintain an *open architecture* that allows for the ‘plugging-in’ or ‘plugging-out’ of components, which includes multiple databases built independently and prior to SEED. For the same reasons, *programming language/schema independence* must be maintained, which influences - among others - the choice of the database system to be used.

Module communication and coordination

The primary method of data exchange utilizes the database; static file exchange cannot provide robust communication or revision management. The database contents evolve over time and keep a record of “interesting” changes. This exchange of design data between modules through the database must obviously be semantically correct and involves always translation (between the database schema and the internal model), if not schema mapping.

Although the modules work with internal design representations that are independent of each other (as opposed to work in a common workspace), they may work on the same database object and modify it differently. These concurrent modifications must be managed properly by the database through an effective transaction mechanism (using our notion of version). In addition, both users and the modules must be able to be informed of object changes.

Available Tools

A general principle followed by us and the other SEED developers is to produce programs ourselves only when they are not commercially available. We want to make use of as much commercial software as possible, while meeting the requirements previously specified. We never use a commercial product if it would compromise the capabilities of the system as a whole. For example, relational databases clearly would not provide the capabilities we need; they were therefore never considered as an option. The next sections identify software capabilities needed by SEED that are commercially available and introduce the software packages we selected in that category.

Object databases

Because modules make use of object-based representations, object database systems have the potential to provide the necessary capability for storing the persistent information generated in SEED. However, not all object database systems are suitable. For example, many object database systems require the use of a specific programming language such as C++, a constraint that directly violates our requirements for programming language independence.

We selected the UniSQL object/relational database system [Kelly *et al.* 1995]. Object/relational databases provide full object implementations, while allowing the incorporation of existing relational databases as a static class within the object system. Additionally, the SQL language is extended to provide complete object management including queries, but not configuration management as described above.

Description logic-based classification software

Description logic is a knowledge representation technique focusing on the identification and multiple classification of concepts. In particular, description logic coupled with an object model provides the ability to define behavior independently of orthogonal classifications. To support this kind of classification, we make use of the CLASSIC knowledge representation system developed by AT&T Bell Labs (recently spun-off as Lucent Technology) [Brachman *et al.* 1991].

Constraint solvers

Both linear and non-linear constraint solving systems have been widely used to solve well-defined optimization problems. We make use of the ILOG Solver constraint solving package to provide these capabilities when needed [Puget 1994].

Geometric modelers

To provide robust geometric modeling capabilities, we chose the ACIS geometric modeling package, which is also used by commercial CAD systems such as AutoCAD [ACIS 1994].

Application Frameworks

Application frameworks provide a software development infrastructure for “well-defined” application areas and user interface capabilities by employing design pattern software engineering principles [Gamma *et al*. 1995]. We use the ET++ public domain application framework that is available on UNIX and Windows platforms. By using this framework, we are able to develop multi-platform applications without platform-specific source code.

Platform-independent run-time systems

Compiled languages such as C/C++ produce programs for a specific hardware platform with the express purpose of execution speed. However, in a heterogeneous software environment, multiple hardware platforms are common and must be accommodated. If traditional development techniques were used, an application would have to be developed for each needed hardware platform leading to extensive redevelopment. Our approach uses the Java virtual machine as a platform-independent run-time system [Lindholm and Yellin 1997]. By employing this kind of run-time system, we can generate programs that will execute on multiple hardware platforms.

The SEED Approach

The above software systems address selected implementation issues, but their functionalities do not converge into a single commercially available system. As a result, we must ourselves provide the integration of these functionalities. Our approach is to define and implement a *modeling environment* that integrates the above capabilities and provides, in addition, the appropriate levels of abstraction needed to build product and process models.

Because such a modeling environment must include both data and behavior specification, we need a platform-independent run-time system that allows us to specify the data management programs once and execute them on a variety of computer and operating systems.

SPROUT

SPROUT (SEED Representation of Processes, Rules and Objects Using Technologies) is a schema-definition language that supports shared schemas from which other representations can be generated [Snyder et al. 1995]. SPROUT provides a robust information modeling environment in which objects, classifications, and attribute values can be defined including those that should be stored persistently.

SPROUT components:

- *Domains* are data-type specifications that establish a legitimate set of values in the same way as database domains do. Domains have a name and an associated set of axioms that constrain the allowable values and are guaranteed to be enforced. Axioms cannot be violated in contrast to *constraints,* which can be violated. Domains are needed because without them, it would be impossible to define, for example, a computable method for translating SI units into English units.
- *Relationship types* establish the behavior for a relationship and have, as a minimum, a name and an associated container (*set*, *vector*, *bag*, or *link*). In addition, one can specify what should be done with objects in the container under the *copy*, *delete*, *anchor*, etc. events (see Section 2). Relationship types also specify the type of association between objects (e.g. one-to-many).
- *Classes* represent an entity or concept of interest that has a computable representation; it is the principal construct used in object-centered representations. A class is defined as a collection of attributes and behavior, some of which may be inherited from a superclass.
- *Classifiers* correspond to the description-logic notion of a *concept*. Multiple classifications can be added to a *class*, which allows to attach multiple classifications to a single *class*.

Role in SEED

SPROUT provides an environment in which not only data, but also behavior can be specified and implemented. A shared SPROUT schema allows module development teams to communicate in terms of the shared information model. It is in fact an important means to arrive at a common understanding of the concepts of mutual interest.

In addition, the schema allows for the semi-automatic generation of several collections of data: We use SPROUT to automatically generate UniSQL database schema descriptions as well as any code necessary to implement the semantics of the information model. Taken as a whole, these components make up part of the SPROUT run-time system.

Because we do not impose the shared information schema on the internal representation of the individual modules, some type of verifiable translation needs to be performed between the database and the module representations. To this end, we provide a *language binding compiler* that, given a specification, will generate code to perform the translation to and from the module to the SPROUT representation (and therefore the database) as illustrated in Figure 3.

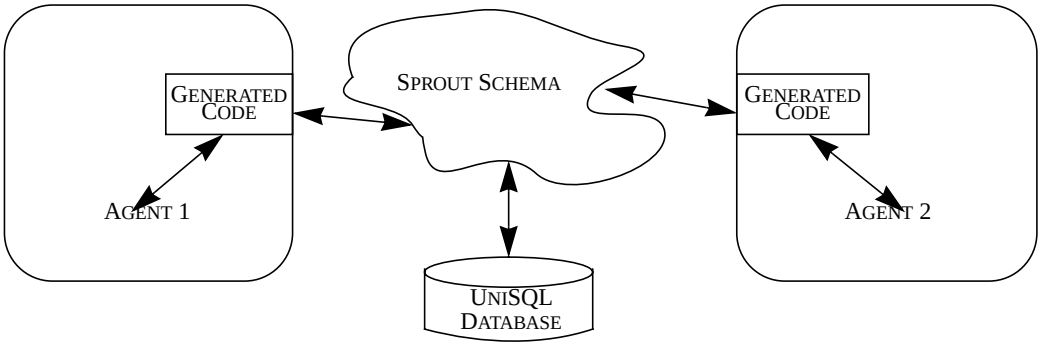


Figure 3 : Shared SPROUT Schema and Language Binding Code

Given this capability, the SEED modules and their users are able to communicate information and to coordinate the use of various object configurations. Changes made to an object by an individual module can be captured as events and be forwarded to the appropriate modules and users.

Conclusion

A database that intends to support a heterogeneous design support environment consisting of independent software modules with diverse internal design models has to satisfy requirements not met by any commercial database system. The design and implementation of this database is an integral part of the overall software engineering effort. The SEED team developed an approach that integrates external and in-house software based on a shared information model specified in the modeling language SPROUT, which allows for the specification of domains, and classes, relationship types and their behavior, and multiple classifications. The SPROUT run-time system organizes and coordinates the communication between the software modules and the database.

Acknowledgments. Work on SEED is currently supported by NSF (through the EDRC), the US Army Corps of Engineers Construction Engineering Laboratory (USACERL), and the University of Adelaide. It involves - aside from the authors - faculty at Carnegie Mellon (O. Akin, S. Fenves, and J. Garrett) and the University of Adelaide (R. Woodbury); and graduate students at both universities.

References

ACIS (1994) **ACIS Documentation**, Spatial Technology, Inc.

Brachman, R. J., McGuinness, D. L., Patel-Schneider, P. F., Resnik, L. A., Borgida, A. (1991) “Living with CLASSIC: When and how to use a KL-ONE-like language” in **Principles of Semantic Networks: Explorations in the Representation of Knowledge** (John Sowa ed.), Morgan Kaufmann Publishers,

Inc., San Mateo, California, 401-456.

Flemming, U. (1994) “Computerunterstützung für den Vorentwurf: das SEED Projekt”, **Wissenschaftliche Zeitschrift**, Hochschule für Architektur und Bauwesen, Weimar-Universität, Jahrgang 40 (1994) Heft 4, 41-48

Flemming, U. (1994a) “Case-based design in the SEED system” in **Knowledge-Based Computer-Aided Architectural Design** (G. Carrara and Y. Kalay, ed.s) New York: Elsevier (1994) 69-91

Flemming, U. and Woodbury, R. (1995) “Software Environment to Support Early Phases in Building Design (SEED): Overview” **Journal of Architectural Engineering**, vol. 1, pp. 147-152

Flemming, U. and Chien, S. F. (1995) “Schematic Layout Design in SEED Environment” **Journal of Architectural Engineering**, vol. 1, pp. 162-169

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) **Design Patterns**, Addison-Wesley, Reading, MA.

Garrett, J. H. Jr., Kiliccote, H., and Choi, B. (1995) “Providing Formal Support for Standards Usage within SEED” **Journal of Architectural Engineering**, vol. 1, pp. 187-194.

Kelly, W., Gala, S., Kim, W., Reyes, T., and Grahm, B. (1995). “Schema architecture of the UniSQL/M multidatabase system.” **Modern database systems: the object model, interoperability, and beyond**, ACM Press, New York, NY, 621-648.

Lindholm, T. and Yellin, F. (1997). **The Java Virtual Machine Specification**, Reading, MA.

Meyer, B (1988) **Object-Oriented Software Construction**, Prentice-Hall, New York, NY.

Puget, J. (1994) “A C++ Implementation of CLP.” In **ILOG Solver Collected Papers**. ILOG Technical Report (URL http://www.ilog.com).

Snyder, J., Aygen, Z., Flemming, U. and Tsai, J. (1995) “SPROUT - A Modeling Language for SEED” **Journal of Architectural Engineering**, vol. 1, pp. 195-203.