Information Sharing in Building Design

James Snyder and Ulrich Flemming
• *Lockheed Martin Advanced Technology Laboratories. Camden, NJ 08102*
• *School of Architecture and Institute for Complex Engineered Systems (ICES), Carnegie Mellon University, Pittsburgh, PA 15213*

Key words: building models, conceptual modeling, product modeling, information modeling, design tool integration

Abstract: We address information modeling and exchange in asynchronous, distributed collaboration between software applications or design agents that are heterogeneous, that is, developed independently based on application-specific data models. We identify the requirements an integration environment must satisfy if it is to support the semantically correct exchange of selected, locally generated information between the agents. These requirements are distilled from both the literature and our own experiments with the Object Modeling Language OML. The resulting requirements were then formalized into an information modeling and exchange environment constructed around the modeling language called SPROUT (supported by a compiler) and an associated software architecture that can be targeted toward many different hardware and software platforms. A unique capability supported in this environment is formal support for integrating existing applications: Given a schematic description in SPROUT, a formal specification can be used to generate computer programs that provably map data to and from the applications.

1. INTRODUCTION

In the present paper, we address conceptual and technical issues that arise in a specific type of collaborative design scenario. We assume that the firms or design teams collaborating on a building project use local software to the degree that is appropriate to them. We assume furthermore that the local software uses algorithms and internal representations/data models tailored to the particular design task it

supports. Each of these local models captures a particular universe of discourse that differs - to some degree at least - from those of the other teams. Each team uses local persistent data storage facilities, i. e. a local database, and it does this again to the degree it deems necessary and appropriate. It is likely that the individual pieces of software were developed independently of each other. It is furthermore to be expected that different teams enter the project at different times, which are, in addition, unpredictable at the outset.

On the other hand, many of the decisions made by one team must be communicated to some of the other teams because they have ramifications for the work done by those teams. Each team intends to take advantage of the fact that it has a computable design representation and to use that representation to automate information exchange to the largest possible degree.

Clearly, our scenario does not deal with synchronous collaboration in which teams work on the same data model or 'document' over some network connection; that is, we do not address situations handled by networked 'groupware' or shared 'electronic whiteboards'. It has been observed that the approaches suggested for a more asynchronous information exchange typically fall into two categories [Rolland and Cauvet 1992] pages 39 and 40: (1) The fully integrated approach, which relies on a unique, shared representational schema to which all subsystems have to conform; and (2) the federated approach, in which each local system has its own schema and is responsible for controlling its interactions with other systems by deciding which information should be imported and exported. With respect to building design, the integrated approach has been described as follows [Augenbroe and Winkelmann 1990]:

> A building project requires generating, updating, and communicating an enormous amount of data... Traditionally this description is stored and displayed in analog, causing numerous problems due to ambiguity, incompleteness and inconsistency of information. There is a strong consensus in the computer industry that the key to integration will be the definition of complete data models for each product type that will satisfy all... information needs. This complete data model is generally called a product model.

Attempts to develop such an approach are too numerous to be listed here. To the best of our knowledge, the federated approach has been tried in the context of building design only in the Agent Communication Language (ACL) project, in which both authors participated [Khedro et al. 1995].

We argue below that neither approach is appropriate in the context of the scenario sketched above, and we introduce subsequently work that attempts to overcome the shortcoming of each approach in this scenario. That is to say, our paper should not be misunderstood as a general critique of fully integrated or strictly federated architectures. Each may well work in different scenarios.

To start with the fully integrated approach in the present scenario: It is unrealistic to expect that the ever-changing cast of teams with their heterogeneous software support can be made to subscribe to a single, comprehensive data model that is able to satisfy the information needs of every participant. In fact, it may not even be desirable to have such a model because individual teams may use data that are of no interest to other teams or even proprietary. The following example, taken from a different industry, illustrates this and shows - at the same time - that this situation is not unique to the building industry.

Figure 1 shows a high-level, simplified decomposition for an airliner that identifies five major subcomponents, each of which is designed by a different team/firm specializing in a specific domain/discipline using internal software, which we call an application below. Information exchange occurs in this situation within and between domain specializations. An example of the former is wing design, which is proprietary, and proprietary data are exchanged freely within the team. An example of the latter is the design of the wing connection to the fuselage, which is contracted out, and information must be exchanged selectively between the different teams responsible for these respective tasks.

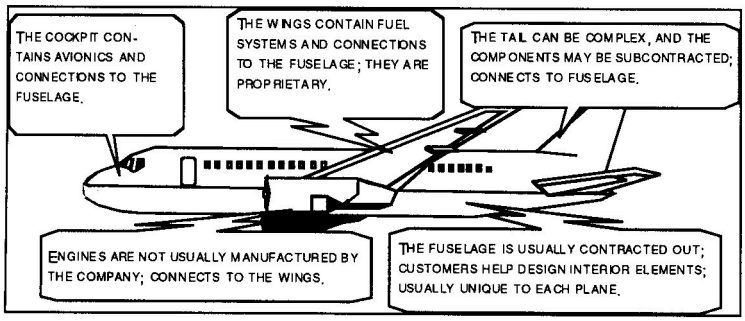

Figure 1. Information Exchange in Airplane Design

We cannot expect that the internal models used by the different applications are conceptually equivalent. An example that arose in the context of the ACL project described in greater detail below should make this clear. Two teams, one representing architectural design, located at Carnegie Mellon University (CMU), and one representing structural design, located at the Massachusetts Institute of Technology (MIT), had to exchange information about spatial configurations and structural systems aimed at supporting these configurations. The MIT team, after receiving an initial spatial configuration from CMU, had to develop a preliminary structural design. It used a very efficient and fully recursive internal representation that made use of the fact that structural members in different locations often are based on the *same* design that satisfies some maximum load condition [Chiou and Logcher 1996]; Figure 2 shows a portion of this representation. The representation therefore recorded the shared design only once and associated it with the different locations where it occurred. The architects at CMU, on the other hand, were not interested in structural details; they were only interested in the geometry of the structural members, which needed to be completely enumerated because they had to be displayed graphically and checked visually. The CMU application therefore represented structural members in a flat list, from where it exported data to a CAD package used only for purposes of 3D display. But it had to remain possible to re-synthesize from this enumerated list particular locations and designs, for example, if the architects did not like the shape or location of a specific element. Note that this type of communication goes beyond the kind of type conversion called transduction in [Eastman 1994]. It requires mappings between the different conceptual schemas involved and may include synthesis and re-synthesis of concepts.

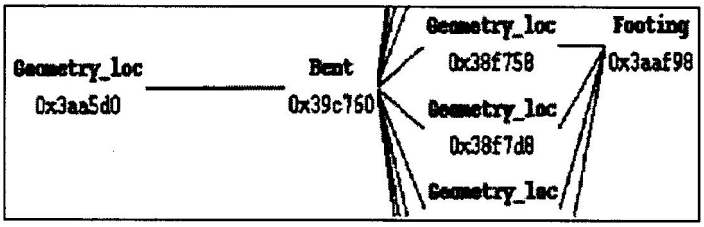

Figure 2. Parsimonious Representation of a Structural System [Chiou and Logcher 1996]

Semantical correctness must be maintained in both directions. More generally, the exchange of (partial) information must satisfy the following criteria under the present scenario:

1. *Robustness of information exchange*: Since applications do not export every piece of information they generate. Information loss will occur during the exchange.
2. *Semantically correct concept mapping*. It must be possible to map bi-directionally information defined in terms of one conceptual schema to information defined in terms of another conceptual schema, and the mapping may require synthesis and re-synthesis of concepts.
3. *Variable rate of information changes*. Some information is only relevant if it is delivered in a timely manner, while other information is relatively static. The information exchange has to be able to take such varying rates into account.

We use the term *communication fidelity* to refer to the three requirements in combination. Note that the example under point 2 above illustrates why a federation architecture as used in the ACL project is inadequate to guarantee communication fidelity as defined here. The facilitator used in this project had no internal memory: it 'flushed out' all information it received immediately to the interested applications; it was thus unable to support the type of synthesis and re-synthesis suggested by the example, which in turn requires that the exchange agent have some form of intermediate representation and internal memory. Indeed, the approach that suggests itself for the present scenario is one that relies on some form of mediated architecture as described in [Wiederhold 1995] in the context of client/server architectures.

We can now state more succinctly the issues addressed in the present paper. We deal with information exchange that maintains communication fidelity for the information exchange in asynchronous, distributed, computer-supported design collaboration as captured in our scenario. Especially, we explore the potential of a mediated architecture that provides not only communication fidelity, but also a large degree of openness and flexibility: agents can enter or leave the collaboration at any time; they are not restricted to a specific universe of discourse or data model, nor to a specific programming paradigm. This scenario raises a host of complex issues that we cannot fully present, even mention, here. We must restrict ourselves to a highly selective presentation; a full account can be found in [Snyder 1998].

The remainder of this paper is structured as follows: We review briefly in the next section research or industrial efforts that influenced our own approach or address similar issues in contrasting manners. The ACL project mentioned above has been especially significant because it provided us with a first test ground for

experiments with a schema specification language, OML. In section 3, we briefly describe the OML experiments and the lessons we learned from them. Both the ACL project and our literature survey convinced us that communication fidelity can best be achieved in an integration environment constructed around a schema specification language; we outline in Section 4 general requirements for such an environment. In section 5, we introduce the schema specification language SPROUT, the successor to OML. In section 6 we outline briefly the kind of integration architecture that can be built with a language like SPROUT and outline a first prototype implementation.

2. BACKGROUND

Computer-Aided Design (CAD) software aimed initially at replacing manual drafting - to some degree - with computer-based tools and was not integrated with any other computer-based design tool. Since then, at least two generations of systems and projects have explored integration issues at increasing levels of generality and sophistication.

2.1 First Generation Integration

A first generation of more integrated systems emerged during the mid-1980's and early 90's. This work concentrated on research prototypes or software-based laboratories and pursued the explicit goal to discover and investigate the issues involved in systems integration in building design. Development on these prototypes has been completed. The following projects were of particular importance to us because of our active participation (see [Snyder 1998] for a more detailed account):

- The Intelligent Computer Aided Design System (ICADS), a project in the Computer Aided Design (CAD) Research Unit, Cal Poly, San Luis Obispo [Pohl et al. 1992] [Myers et al. 1992]
- The Integrated Building Design Environment (IBDE) project in the former Engineering Design Research Center (EDRC) at Carnegie Mellon University [Fenves et al. 1994]

2.2 Second Generation Integration

Second generation integration systems make use of results from first generation systems and benefit generally from a significantly larger suite of commercial or research tools that could be used as subsystems of the large whole (e.g. objectoriented databases). The projects presented below represent solutions to the information exchange problem that are - within their own premises - persuasive and successful.

2.2.1 STEP and STEP-based Extensions

The Standard for Exchange of Product Model Data (STEP) (ISO 10303) is an ISO committee with the purpose of developing an infrastructure for product data

exchange across disciplines. The Product Data Exchange Using STEP (PDES) project is a U.S. project that provides industrial input to the ISO committee [Nell 1996]. In order to facilitate the definition of product data models, the EXPRESS language continues to evolve [ISO 10303 Part 11]. Both STEP and EXPRESS are quite large, and we cannot provide a comprehensive review in the context of this paper.

EXPRESS was initially designed to facilitate file-based data exchange using a neutral file format and provides several syntactic mechanisms to support this type of exchange [Eastman and Fereshetian 1994]. EXPRESS's usefulness has been demonstrated in several application areas. It is at its core an instantiation specification language, not a schema specification language (see [Snyder 1998]).

Among the STEP-based extensions, the COMBINE project is of particular interest to us because it reinforces insights from the ACL project, which revealed severe problems associated with schema mappings and translations that must be coded manually [Dubois et al. 1995].

2.2.2 DICE

The Distributed and Integrated Environment for Computer Aided Engineering (DICE) project is a broad research effort of the Intelligent Engineering Systems Laboratory at MIT [Sriram and Logcher 1993]. DICE is a network of collaborating design agents, where communication and coordination are achieved through a global database and control mechanism. The product and processing modeling component of DICE is SHARED [Wong and Sriram 1994a]. Again, DICE is too extensive to be reviewed here in full; we must restrict ourselves to a few comments about shared, the most interesting component of DICE in the present context.

SHARED can be described as a rich object-oriented modeling environment. It also provides database capabilities such as version, alternative and transaction management. [Sriram et al. 1994] view the SHARED approach "... as a first cut in developing a semantic vocabulary for design, where the constraints (through behavior) implicitly define the grammar for checking valid designs." In its current form, SHARED should be viewed an as implementation framework for shared workspaces and not a conceptual modeling environment because concepts are defined in the implementation language, not in a programming language-independent way.

2.2.3 EDM

The Engineering Data Model (EDM) is a modeling environment [Eastman and Siabiris 1995]. EDM-2 [Eastman et al. 1995] added object structuring techniques in terms of a specialization lattice and composition hierarchy. Both constructs are general and have direct analogies to other object paradigms, in particular object/relational database systems, that support schema evolution. EDM-2 can be viewed as a database schema definition language with the added capability of constraint definition and management of the objects. Among 2nd generation integration projects, EDM is in spirit closest to our own work because it explicitly addresses the scenario outlined in Section 1.

2.3 Context

Starting in 1992, groups at the School of Architecture and Department of Civil and Environmental Engineering at Carnegie Mellon University (CMU) have been developing the Software Environment to Support Early Phases in Building Design (SEED) with participation of a group at the University of Adelaide, Australia [Flemming and Woodbury 1995]. SEED builds upon the experience gained with the IBDE project and several projects dealing with 'generative' design systems in the EDRC.

The SEED project intends to develop the prototype of a software environment to support 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 (i.e. *design tools*), 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 SEED team specifically develops generative software components, called modules, each as it relates to a specific discipline or application domain. Design tools are to be provided to the largest possible degree by third-party packages. SEED is to be used is exactly as described in Section l's senario, except that the individual modules rely on a shared object server for persistent data storage and version management and share basic design concepts and are not required to use a specific programming language. The environment has to remain heterogeneous and open simply because the SEED developers cannot predict which software has to be integrated in the future and when.

3. OML EXPERIMENTS

We describe in this section experiments with a high-level specification language, called the Object Modeling Language (OML), that were conducted in the context of the ACL project mentioned in Section 1

3.1 ACL Project

The multi-institutional ACL project aimed at investigating concurrent engineering (CE) technologies to support collaboration between participants in the facility design and delivery process [Khedro et al. 1995]. Teams of researchers representing four major universities participated along with a team at the U. S. Army Construction Engineering Research Lab (USACERL), the project sponsor: MIT, Stanford University, the University of Illinois at Urbana-Champaign, and CMU. The collaboration between the teams was to make use of technologies and applications developed independently by the four universities and USACERL. The CMU team was to use SEED-Layout, a module of SEED, to deal with the architectural aspects of early building design [Flemming and Chien 1995].

The sponsor of the ACL project had established at the outset that the cooperation of the cooperating systems was to be based on the agent-based Federation Architecture developed by Stanford, which intends to allow independent software agents to exchange information by communicating with facilitators through appropriately designed APIs [Khedro et al. 1995]. The ACL project it is the only project known to us that addresses the type of asynchronous, distributed collaboration in building design underlying our scenario through a strictly federated architecture that had to be implemented in order to give researchers the opportunity to investigate the more general problems posed by this scenario through realistic experiments, that is, experiments using applications that were truly distributed as implied by the scenario. More specifically, this project lead to the OML experiments that proved so crucial for the present work.

3.2 Object Modeling Language

During the course of the project, the team members realized that because each team had its own internal agent architecture, some kind of unifying syntax and semantics of object descriptions was needed to allow message exchanges to occur. The CMU team developed OML as one of its contributions to the ACL project. OML provides an object description schema with execution semantics, a messaging protocol, and an associated language binding complier that generates data structure mapping code from an OML schema, that is, formal specification [Snyder and Flemming 1994a], [Snyder and Flemming 1994b]. Based on an agent's internal representation, an OML schema builds an OML model external to the agent, which describes the agent's model to the outside world in terms of four basic language constructs: domains, relationship types, and classes (a complete description of OML is given in [Snyder 1993]). OML was designed to provide semantic consistency for inter-agent information exchange.

Given an OML schema, each agent developer must provide a (concise) language binding specification that can be used to map the local representation to and from the OML schema. The mapping between different local representations involves a large amount of monotonous binding assignments, which, in addition, must be verified. This verification is error-prone and time-consuming, which is especially true when the mapping routines need several programming language linkages (e.g. C, C++, Fortran). [Dubois et al. 1995] make the same observation (see also {Lockley et al. 1995]); IDM stands for Integrated Data Model and *DT* stands for Design Tool:

> For the time being, too much is still dependent on individual skills and inspiration. Proofs that a resulting model is somewhere near a desired optimum or even semantically correct are still outside of our reach. A particular aspect of the latter is the way that teams now have to check whether their DT model can be mapped to and from (a particular subschema of) the IDM. This in fact is the way that the IDM was checked and refined in the latter stages of the project. As it happens there is no other way available than to try to perform the mapping for a range of examples given the IDM and the DT's aspect model. This is a time consuming task which moreover can lead to a false conclusion depending on the completeness of the chosen examples.

The problem is exacerbated when the schema itself is not entirely static, for example, because the schemas underlying the individual applications are still

evolving or new applications are added to the environment. Reconfiguring the schema in these cases may be practically impossible.

These problems are largely circumvented by the type of language binding compiler provided in the OML standard distribution: The developers can use the OML schema and the language binding specification to generate automatically language binding code in a host language like C++. This code can be compiled into a language binding library and linked to an agent's domain code along with OML run-time libraries.

3.3 Observations

OML was tested in two contexts, the ACL project for which it was designed, and the Modular Design System (MDS) project, another USACERL-funded project (not presented here). The language binding compiler was tested both internally in the communication between SEED-Layout and the other agents communicating with SEED-Layout to form the architectural agent in the ACL project (an object database and CAD package used strictly for 3D displays). It was also tested in information exchange between the CMU- and MIT- based agents. But these experiments were conducted in parallel to the main project focus.

The system as a whole was never complete or tested. The Federation Architecture was not able to support the integration of applications of the given type over the Internet for several reasons, among which the following are most significant in the present context: (1) We never knew if information was exchanged correctly because we could not verify concept translation rules or data structure mappings in an application; translations must be based on a formalisms and automated code generation. (2) The Federation Architecture did not provide any tools for object-centered information modeling. Some researchers claim that it should not have such tools, but became clear to us that state-of-the-art object-based modeling techniques, such tools are essential. [Flemming et al. 1996] provide a full report of the research results of the ACL project from the SEED perspective; see [Chiou and Logcher 1996] for a complementary assessment.

Combined with the OML experiments, the results of this project strongly suggest that a robust, programming language-independent conceptual modeling language is needed for information exchange techniques to be scalable. Specifically, the language binding compiler demonstrates a promising approach for generating mapping code from a formal specification rather than by the current techniques that rely on code generated by hand.

4. REQUIREMENTS FOR AN INFORMATION MODELING AND EXCHANGE ENVIRONMENT

The reviewed literature and OML experiments provided us with a wealth of insights and suggestions about promising ways to deal with the information exchange issues raised in our scenario. In the present section, we distill from these a set of requirements an information modeling and exchange environment should meet

in this context. The requirements are somewhat motivated by the SEED perspective. They remain however general because SEED is conceived in general terms.

4.1 Modeling Language

The cornerstone of the environment is a modeling language. Before we elucidate the requirements such a language should meet, we provide a brief overview of modeling languages as an emerging research area.

The complexities associated with developing large-scale information systems have pushed existing programing languages beyond their capabilities. This has led to the introduction of high-level modeling languages that can be used to model functional application requirements and information system components at a conceptual level [Loucopoulos 1992]. Given the different areas from which modeling languages historically evolved (knowledge representation in Al, programming languages, data bases), it is difficult provide a concise definition that would cover all instances because different languages tend to retain some of the flavor of the field in which they originated. But certain common traits do emerge.

The most unifying force is an object-centered approach to representation. The most practical way to distinguish modeling from programming languages is to describe the features incorporated in the language as opposed to what constructs can be represented in it (which may have the same names). Several such distinguishing features have emerged in the literature:

- **Semantic Relationships**. Object-oriented programming languages have no explicit constructs for representing the semantics of object relationships; that is, complex object compositions must be maintained manually by the programmer. However, some modeling languages have the ability to manage object compositions using relationships between objects as an explicit construct in the language. Therefore, a more consistent and uniform mechanism can be employed to address relationship management. In fact, several authors suggest that relationships can be generalized with their own properties and definitions ([Rumbaugh 1987], [Parent and Spaccapeitra 1992]).
- **Behavioral Properties**. In addition to the more traditional encapsulation capabilities found in programming languages, modeling languages allow for the specification of reactions to events that occur on an object. To borrow from database terminology, an event can trigger an action. Other forms of behavioral properties include constraints on the structural or data properties of an object.
- **Dynamic and Temporal Capabilities**. Modeling languages handle changes to objects over time while maintaining some type of historical record of these changes; in database terminology, this is usually version and configuration management. In some cases, semantic relationships and configuration management are highly interconnected (see below).
- **Representational Homogeneity**. Object-centered representations have the desirable property of using similar representations at different levels of conceptual abstraction. Traditionally, product models were specified and represented independently from the process models that utilized them. However, a homogeneous representation should support both simultaneously.

4.2 Overall Schema Requirements

The modeling language supporting the application integration environment envisaged is used to formally specify a conceptual schema serving the following functions:

1. Provide a 'lingua franca' used by participating applications for mappings between concepts of mutual interest. The OML experiments clearly demonstrate the power and flexibility gained from basing information communication on such an exchange medium. The modeling language is used to specify a shared schema to and from which the schemas underlying the participating applications are mapped through language bindings.
2. Support the semi-automatic generation of language binding code. Again, the OML experiments, supported by the literature, demonstrate the feasibility, indeed, the necessity of this approach. In the SEED context, this includes also the automatic generation of the database schema from a high-level specification.
3. Provide basis for negotiations between application developers about the shared concepts.

The last point needs some elaboration because it is universally neglected in the literature. We can assume that at the outset, only the developers of the individual applications understand the semantics their underlying the data models. In order to develop a shared schema, the individual developers have to engage in rounds of negotiations in which they 'hammer out' both the content and form of that schema. The OML experience as well SEED-internal negotiations suggest to us that the modeling language itself can provide crucial support for this process by establishing from the outset a common communication medium. Indeed, the language can provide a firm structure for the negotiations.

For example, the following process proved successful for negotiations between the SEED teams. Before the first meeting, every team has to do its 'homework', that is, express the shareable aspects its own internal data model in terms of the modeling language (as a side effect, this puts the expressive power of the language to the test). These specifications serve to explain to each team the content of the other teams' models. Next, each team indicates which pieces of information it is interested in receiving from the other teams thereby determining both the content of the shared schema and the information that can safely be lost. The group as a whole is then ready to design the shared schema. This process can take advantage of the fact that the information of interest is already expressed in the target language; that is, it consists largely of remixing what is already specified. Central to this last, all-important step is to decide where information loss occurs. The mediated architecture envisaged here gives developers flexibility in this process.

If a new application enters the cooperation, this same process is repeated at a smaller scale. In all likelihood, the shared schema has to be updated as a result and the language bindings recompiled, a process that is largely automated if the schema is conceived along the lines envisioned here.

We conclude this section with some general schema requirements:

1. **Implementation Independence**. One of the more challenging requirements information modeling languages try to meet is the elimination of hardware and software platform dependencies. While the success in this effort must be

achieved at various levels of abstraction and with various resources, particular attention must be paid to schematic features that may diminish platform independence. For example, something as simple as upper and lower case names within a model can make implementation difficult or impossible in some environments. Additionally, the modeling environment should be independent of a programming language.

2. **Application Integration** must be supported by formal methods that allow for the (semi-)automatic generation of language binding code so that any changes to the schema require only recompilation. Specifically, given a shared schema, an application binding specification should be used to generate code to perform the mapping.

4.3 Schema Elements

During the OML experiments, we discovered that the representations used to specify the mentioned above mappings should reflect an object-oriented structure if some applications use object-based representations. Both the OML experiments and the SEED experience suggest that an object-centered modeling language needs at least the following constructs to gain the needed expressiveness:

- **Domains**: An important distinction is the differentiation between well-formed data and representations of design objects of interest. 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 and 90 degrees). Consider, on the other hand, a structural engineer who starts with the design of the roof structure for a building and abandons work for the day without designing any of the support elements. Clearly, the structural system is physically ill-formed, but the engineer should still be able to communicate this incomplete design to others. The sun ray information represents data, whereas the structural system is a design object of interest. A schema should be able to enforce well-formedness of data in the above sense; it has no business examining design data beyond that.

  The most appropriate formal representation for well-formed data is abstract data types (ADTs) as defined by [Meyer 1988]. But modeling language ADT implementations are different from programming languages. Preconditions and postconditions can be offered in both kinds of languages, but the modeling languages require additional support that is almost never provided by programming languages. For example, it should be impossible to describe nonsensical operations on dimensions measured in a unit system, for example, the multiplication of two volumes. To make the distinction between ADTs in the different languages clear, we call the representation of a data type a domain in modeling languages. Domains may support inheritance, but domain-based inheritance is essentially different from class-based inheritance (discussed in [Snyder 1998]).

- **Relationship Types**: A relationship type represents a kind of association that can exist between design objects; if it has an inverse, it has to be managed by the schema. They may need associated behavior of the association under the significant data base events described as described below. We decided not to

model relationship types as classes for both practical and conceptual reasons.

- **Classes**: Design objects of interest, which differ from well-formed data, are instances of classes within an information modeling language. A class can be defined as an aggregation of attributes of different types that incorporate domains and relationship types. Classes must support inheritance in the more traditional sense, but should support multiple-inheritance for reasons given below.

- **Multiple Classifications**: Clearly, objects of interest need to be multiply classified, for example, to support various queries both during evaluations (e.g. is this space public so that a pathfinding algorithm can treat is as node). Multiple inheritance is many times inappropriately used to classify objects having nothing or very little to do with the behavioral aspects of a class (see [Kiloccote 1996]). A classification scheme should support operations such as subsumption and the detection of conflicts. The work most relevant to SEED is the CLASSIC knowledge representation system, not the least because it incorporates algorithms that are known to be tractable [Brachman et al. 1991].

4.4 Database Support

To overcome the limitations of relational databases, researchers have proposed various approaches able to support an information modeling environment that supports object-centered representations, which we cannot review in detail (see [Snyder 1998]). We point out a specific aspect required by a system like SEED, which is capable of generating large numbers of alternative designs that can undergo version changes, where the designs are modeled as complex object configurations, some of which need to be stored persistently.

Relationships between objects in a configuration behave differently under different database events and must be managed appropriately. Take again the structural system from Figure 2 as an example. An alternative version may use the same designs, which are already stored in the database, but in different locations. When storing the new version, the database has to know that the locations have to be created, but not the associated design, and be able to link correctly, all objects, new or existing. Conversely, when a design is deleted, all associated locations have to be deleted, too. We call this configuration management.

In general, the SEED database must be able to manage properly the following database events:

1. **Delete**: When an object in a configuration is deleted, the database must know which of the associated objects should also be automatically deleted.
2. **Copy**: Copying a configuration means the construction of a complete, 'deep' copy. Usually all objects in the configuration are copied and the copies linked in a network that is isomorphic to the original but exceptions must be specifiable.
3. **Anchor**: Anchoring a configuration means creating a different version of a configuration that can be retrieved based on a current time stamp. In this case, configuration management depends entirely on how a given schema is defined.
4. **Overwrite**: Overwrite replaces a configuration in the database with a configuration that has the same identifier, but a new time stamp. Overwrite then proceeds similarly to anchor.

If a schema is able to distinguish relationship types by the way in which they behave under these events, configuration can be automated based on these specifications.

5. THE SPROUT INFORMATION MODELING LANGUAGE

We briefly describe in this section the SPROUT (SEED representation of Processes, Rules, and Objects Utilizing Technologies) modeling language. The overall objective of SPROUT is to provide a syntactic and semantic framework that embodies the requirements defined in the preceding section. This section introduces the schematic language elements in SPROUT depicted in Figure 3. The issues surrounding domain and class schematic elements were introduced in Section 4. The notion of a unit of measure is a relatively unique modeling language concept that we cannot further discuss here. SPROUT specifications are detailed in [Snyder 1998].

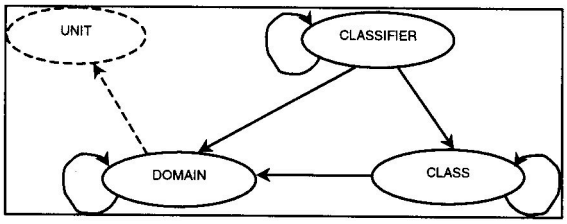

Figure 3. Relationships Between SPROUT Schematic Elements

The SPROUT syntax is heavily influenced by the Eiffel object-oriented language [Meyer 1988]. Eiffel embodies good language design principles and has the added benefit of having abstract data types at the core of its language design including support for exceptions due to invariant violations. It was therefore a natural and important reference work. It is important to note however, that the overall software architecture and semantic assumptions of SPROUT differ from Eiffel because Eiffel is a programming, not a modeling language.

Unlike Eiffel, SPROUT supports qualified schematic element naming via a package naming, thereby defining a package scope in a manner similar to, but not identical with the Java programming language [Gosling et al. 1996]. Package naming is an essential feature in the definition of shared schemas. Without this construct, it becomes virtually impossible to manage element naming within a SPROUT schema (or any schema language for that matter). Something as simple as a two-dimensional point can be defined in several different contexts, and without package names, there would almost certainly be collisions between the names.

5.1.1 Domains

Domains support the notions described in Section 4.3. To give readers a flavor of the SPROUT syntax, Figure 4 shows the SPROUT specification of a simple

domain, Meter (sprout.lang is the default package name for built-in elements) that is based on built-in domains. The elements of a complex domain must refer to an existing domain to make up structured or recursive domain definitions. The complex domains found in SPROUT are *sets, sequences, ordered sets, ordered sequences, tuples*, and *constraint variables*.

```
forward domain SquareFeet, SquareMeter specializes sprout.lang.Real
  implements Area end

domain Meter specializes sprout.lang.Real implements basecase Length
  operator Meter ^ Meter: error is default end
  operator Meter * Meter: SquareMeter is default end
  operator Meter / Meter: sprout.lang.UnitlessReal is default end
  operator Meter < Meter: sprout.lang.Boolean is default end
end
```

Figure 4. Example Simple Domains

SPROUT also supports domain transformation capabilities called *function*, *operator*, and *conversion*. These transforms, in particular function, resemble methods found in object-oriented programming with some important difference in behavior. In particular, inheritance plays a significantly different role.

5.1.2 Relationship Types

Relationship types establish a behavior for a relationship and have, as a minimum, a name and an associated container (i.e. *set*, *vector* or *bag*, and *link*). Additional information can be specified in a relationship type, for example, what should be done with objects in the container under the database events introduced in the preceding section.

5.1.3 Classes

Classes are defined as collections of attributes. Due to space limitations, a complete definition is not provided here (see [Snyder 1998]), rather the kinds of attributes are simply stated. In SPROUT, attributes have the following types: value, relationship, method, trigger, derived, and constraint. An example SPROUT class is shown in Figure 5.

```
class Point specializes sprout.lang.SproutObject
      value x sprout.lang.UnitlessReal;
      value y sprout.lang.UnitlessReal;
end

class Rectangle specializes sprout.lang.SproutObject
  value length Meter default 10.0;
  value width Meter;
  relationship lowCorner hasOne(jim.Point);
  relationship highCorner hasOne(jim.Point);
  method m: Meter is
    local
      p: jim.Point;
    do
       width:= 10.0;
       p.Create;
       p.x:= 10.0;
       p.y:= 20.0;
       lowCorner.assign(p);
    end
  derived area     SquareMeter is length * width;
end
```

Figure 5. Example Class Specifications

5.1.4 Classification

During the course of this research, it became clear that, for implementation reasons, support for classification was better developed in the context of the case-based components of the SEED system. As a result, plans to include a formal classification specification in SPROUT were abandoned. However, it is important to include clear and specific definitions of what classification means in the context of an information modeling language.

5.2 Implementation Prototype

At the time of this writing, we are still working on a first implementation of the modeling and exchange environment envisioned here. We describe in the present section briefly the main components of this first implementation.

In order to assure platform independence, the initial middleware server environment is implemented in the JavaServer toolkit. Figure 6 shows the overall process of compiling a SPROUT schema into this type of environment. Note the differences between the runtime and compile-time relationships.

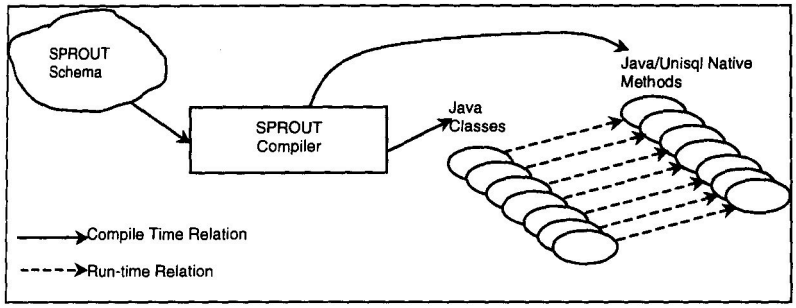

Figure 6. SPROUT Schema Compilation

The specific outputs generated using this target environment consist of a collection of public Java class files and an associated native method implementation file. These Java classes are specializations of Java classes developed as part of the SPROUT runtime environment. Using this technique, many behaviors can be properly implemented without any intervention on the part of the compiler back-end developers or the schema designers. Additionally, these classes are installed as part of a middleware server environment not described here.

ACKNOWLEDGEMENTS

The work reported here has been supported by the US Army Corps of Engineers Construction Engineering Research Lab and, until 1996, by the Engineering Design Research Center at CMU.

REFERENCES

Augenbroe, G. and F. Winkelmann (1990). *Integration of Simulation Into the Building Design Process*. TU Delft Research Report, TU Delft, Holland.

Brachman, R. J., McGuiness, D. L., Patel-Schneider, P. F., and L. A. Resnick. (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* J. Sowa (ed.) San Mateo: Morgan Kaufmann.

Chiou, J.-D. and Logcher, R. D. (1996) *Final report. Testing a Federation Architecture in Collaborative Design Process*. Research Report R96-Ol. Department of Civil and Environmental Engineering. Massachusetts Institute of Technology. Cambridge, MA.

Dubois, A. M., Flynn, J., Verheof, M. H. G., and G. L. M. Augenbroe (1995). "Conceptual Modelling Approaches in the combine Project" in *Proceedings of ECPPM '94 - The First European Conference on Product and Process Modelling in the Building Industry* R. J. Scherer (ed.), Dresden, Germany, October 5-7, 1994, Rotterdam: A. A. Balkema.

Eastman, CM. (1994), "A Data Model for Design Knowledge" in *Knowledge-Based Computer-Aided Architectural Design* G. Carrera and Y. E. Kalay (eds.), Elsevier Science BY., Amsterdam, The Netherlands. pp.95-122.

Eastman, C. M. and N. Fereshetian (1994). "Information models for use in product design: a comparison" *Computer-Aided Design* 26, pp. 551-572

Eastman, C. M. and A. Siabiris (1995). "A Generic Building Product Model Incorporating Building Type Information" *Automation in Construction*, 4(4), pp. 283-304.

Eastman, C., Cho, M. S., Jeng, T.S., and H. H. Assal (1995). "A Data Model and Database Supporting Integrity Management" in *Computing in Civil Engineering* J. P. Mohsen (ed), volume I of 2, pp. 517-524, Proceedings of the Second Congress held in conjunction with the A/E/C Systems 95, Atlanta, GA, June 5-8, New York: American Society of Civil Engineers.

Fenves, S., Flemming, U., Hendrickson, C.,Maher, M. L.,Quadrel, R.,Terk, M., and R. Woodbury (1994). *Concurrent Computer-Aided Building Design*, Englewood Cliffs, NJ: Prentice-Hall.

Flemming, U. and Chien, S. F. (1995). "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering*, 121(4), New York: American Society of Civil Engineers. pp. 162-169.

Flemming, U. and R. Woodbury (1995). "Software Environment to Support Early Phases in Building Design (SEED): Overview" *Journal of Architectural Engineering*, 121(4), New York: American Society of Civil Engineers. pp. 147-152.

Flemming, U., Aygen, Z., Snyder, J. and J. Tsai (1996). *A2: An Architectural Agent in a Collaborative Engineering Environment*. Technical Report EDRC-48-38-96, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.

Gosling, J. Joy, B., and G. Steele (1996). *The Java Language Specification*, Reading: MA

ISO 10303 Committee. Part 11 - *The EXPRESS Language and Reference Manual*.

[Khedro, T., Case, M. P., Flemming,U., Genesereth, M. R., Logcher, R., Pedersen, C., Snyder, J.,Sriram, R. D. and P. M. Teicholz (1995). "Development of a Multi-Institutional Testbed for Collaborative Facility Engineering Infrastructure" in J. P. Mohsen (Ed.) *Computing in Civil Engineering: volume 2: Proceedings of the Second Congress held in conjunction with the A/E/C Systems '95*, Atlanta, GA, June 5-8, New York: American Society of Civil Engineers. pp. 1308-1315.

Kiliccote, H. (1996). *A Standards Processing Framework* Ph.D. Thesis. Dept.of Civil and Environmental Engineering. Carnegie Mellon University, Pittsburgh, PA.

Lockley, SR., W. Rombouts, W. Plokker (1995). "The COMBINE Data Exchange System" in *Proceedings of ECPPM '94 - The First European Conference on Product and Process Modelling in the Building Industry* R. J. Scherer (ed.), Dresden, Germany, October 5-7, 1994, Rotterdam: A. A. Balkema.

Loucopoulos, P. (1992). "Introduction: Section One: Conceptual Modeling" in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information System Development* P. Loucopoulos and R. Zicari (eds). New York: John Wiley & Sons, Inc. pp. 1-26.

Meyer, B. (1988). *Object Oriented Software Construction*. Prentice-Hall, Englewood-Cliffs, NJ.

Myers, L., Snyder, J. and L. Chirica (1992). "Database Usage in a Knowledgebase Environment for Building Design" *Building and Environment*, 27(2), pp.23 1-241.

Nell, J. (1996). "Step on a Page" Technical Report. National Institute of Standards and Technology. Gaithersburg, MD. (see STEP Web site at http://www.nist.gov/sc4).

Parent, C. and S. Spaccapierta (1992). "ERC+: An Object-Based Entity Relationship Approach" in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information System Development* P. Loucopoulos and R. Zicari (eds). New York: John Wiley & Sons, Inc. pp. 69-86.

Pohl J., L. Myers, J. Cotton, A. Chapman, J. Snyder, H. Chauvet, K. Pohl, J. La Porta (1992). *A Computer-Based Design Environment: Implemented and Planned Extensions of the*

ICADS Model. Technical Report, CADRU-06-92, CAD Research Center, Design Institute, Cal Poly, San Luis Obispo, CA, 93407.

Rolland, C. and C. Cauvet (1992). "Trends and Perspectives in Conceptual Modeling" in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information System Development* P. Loucopoulos and R. Zicari (eds). New York: John Wiley & Sons, Inc. pp. 27-48.

Rumbaugh, J. (1987). "Relations as Semantic Constructs in an Object-Oriented Language" OOPSLA '87 as *ACM SIGPLAN22*(12). pp. 466-481.

Snyder, J. (1993). A Semantic Modeling System for CAD, M.S. Thesis, Department of Architecture, Cal Poly, San Luis Obispo, CA.

Snyder, J. D. (1998) *Conceptual Modeling and Application Integration in CAD: The Essential Elements*. Ph. D. thesis, School of Architecture, Carnegie Mellon University., Pittsburgh, PA.

Snyder, J. and U. Flemming (I 994a) *The Object Modeling Language (OML) Specification*. Internal Technical Report (URL: http://seed.edrc.cmu.edu/ACL).

Snyder, J. and U. Flemming (1994b). The Object Modeling Language (OML) Language Binding Specification. Internal Technical Report (URL: http://seed.edrc.cmu.edu/ACL).

Sriram, D. and R. Logcher (1993) "The MIT Dice Project" *IEEE Computer* (Jan), pp. 64-65.

Sriram, D., Logcher, R., and A. Wong (1994). "Computer Supported Collaborative Engineering: Research Issues in Product Modeling" in *Bridging the Generations: International Workshop on the Future Directions of Computer-Aided Engineering* (position papers submitted to the workshop) D. Rehak (ed.), July 18-19, Carnegie Mellon University, Pittsburgh, PA.

Wiederhold, G. (1995). "Mediation in Information Systems" *ACM Computing Surveys*, 27(2), New York: ACM Press, pp. 265-267.

Wong, A. and D. Sriram (1994). *Shared Workspaces for Computer-Aided Collaborative Engineering*. Technical Report. Intelligent Engineering Systems Laboratory, Dept. of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA.