IMPLEMENTATION OF A DYNAMIC INFORMATION SYSTEM FOR DESIGN

An information system for design

S. FRIDQVIST
Computer Aided Architectural Design, Lund University, Lund, Sweden

Durability of Building Materials and Components 8. (1999) *Edited by M.A. Lacasse and D.J. Vanier*. Institute for Research in Construction, Ottawa ON, K1A 0R6, Canada, pp. 2569-2578.

Abstract

This paper reports an implementation of the conceptually most important features of the BAS•CAAD information system, and the use of this implementation to create models of different levels of generalisation in the construction context. The foundations for the BAS•CAAD information system, which have been presented in an earlier paper, are briefly described. It is a dynamic information system for design, built on a generic ontological framework. The system supports the definition of classes in different levels of universality; the classes may originate from different standards or the individual designer, and allows a free combination of attributes.

Keywords: CAD, design, dynamic schema evolution, information systems, object oriented modelling, product modelling

1 Introduction

Many approaches of product modelling focus only at modelling, and seem to overlook the process of creating the models. The most outstanding feature of this process is that the information changes and evolves over time, not only in quantity but as well semantically. This would make it hard to use a product modelling system based on a fixed classification schema in the earliest, most dynamic phases of design, since the fixed schema would be at odds with the evolving semantics of design.

This paper presents the author's work of implementing a prototype information system, which has been constructed from the basis of the theoretical investigations of the BAS•CAAD research project. The project aims to find solutions to both the problem of modelling, and to the need to reflect and support the evolving nature of the design process. The theoretical foundation of the project was initially developed by Dr. Anders Ekholm, and has been refined by him and the present author as part of the BAS•CAAD project.

1.1 Information systems for design

Information systems are computer based systems that support handling information, e.g. during problem solving. A more thorough discussion of information systems for design can be found in (Ekholm and Fridqvist 1998). There, we defined that information systems for design must: 1) support representing objects in the domain of interest, 2) be able to communicate with other software, 3) support defining the design goal, and 4) have a dynamic object structure.

Most proposals for product modelling software focus on the first two requirements. To be useful not only for describing the results of the design process, but as a tool in this process, an information system for design must have all properties mentioned above. This is discussed in e.g. Eastman and Fereshetian (1994), Eastman, Assal and Jeng (1995), Galle (1995), Junge, Steinmann and Beetz (1997), Leeuwen and Wagter (1998), and Ekholm and Fridqvist (1998).

2 Foundations for the BAS•CAAD information system

The BAS•CAAD information system is intended to cover all levels of generalisation of modelling in the construction context, from international classification standards to specific buildings. It is built on a generic ontological framework, with the object classes *thing class*, *relation* and *unary attribute*, and it supports generic design operations, like generalising and specialising, aggregating and decomposing, and adding and removing attributes (Fridqvist and Ekholm 1996).

The BAS•CAAD ontological framework allows models to be multi-contextual; that is, several contextual views can be co-ordinated in one model. The foundation for multi-contextuality is aspect views, as described in (Ekholm and Fridqvist 1998). However, although multi-contextuality is inherent in the BAS•CAAD object structure, it has yet to be implemented as an observable feature in the user interface.

In

Fig. 1 the ontological framework is depicted as an EXPRESS-G diagram. The figure differs from the one in (Ekholm and Fridqvist 1998) in that the entity Attribute is now named UnaryAttribute. In addition, the superclass BAS_CAAD object is removed.

The BAS•CAAD information system is based on a similar object oriented view as object oriented programming languages. In its present state, however, the BAS•CAAD information system does not support definition of the objects' behaviours (see section 3.1). For simulation purposes, a modelling system must enable defining the behaviour of the objects. The present development of the BAS•CAAD information system is intended to show how a system that describes classes of concrete things can support design; simulation modelling is not supported, although such an extension is possible.

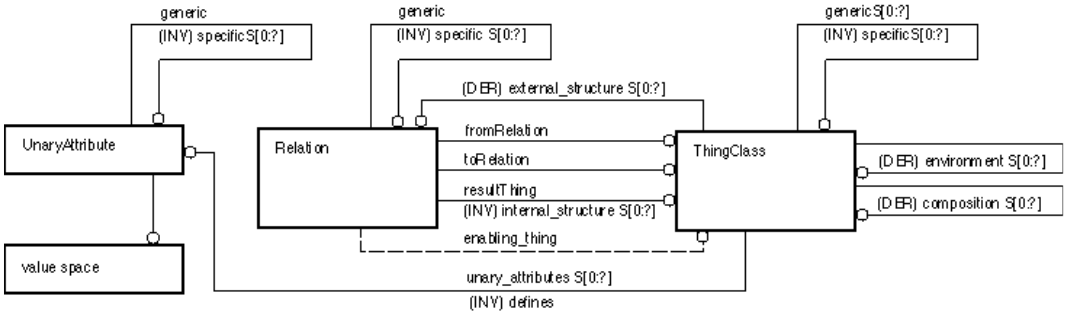

Fig. 1: Ontological framework of the BAS•CAAD information system

2.1 Design statements and attributes

Artefact design can be characterised as making statements about the designed thing and the 'satisfactory situation' that the design is to support (Ekholm and Fridqvist 1998). An information system for design should accordingly at least be able to record such statements. The statements, of course, need not be verbal. Also 2D drawings or 3D models or parts thereof can be treated as statements. The BAS•CAAD information system is designed to collect such statements and to support analysis and communication of them.

Note that not all designer actions result in new statements in the meaning used here. Some actions change or delete already present statements. Thus, statements are not identical to the moves Schön discussed. Schön used the term *move* for any physical or mental activity the designer takes regarding the design task (Schön 1983).

The attributes of the entity ThingClass in Fig. 1reflects that it can be defined as a 6-tuple of sets of attributes, T = (T_G, T_C, R_I, T_E, R_E, A_U), (Ekholm and Fridqvist 1998). Given the BAS•CAAD ontological framework, only seven different types of statements are possible. They correspond to creating ThingClasses, and to adding attributes to the six sets mentioned above:

1. There is a kind of things, called X.
2. An X-thing is a kind of Y-thing (= kind X is a subkind or specialisation of Y).
3. An X-thing is composed by a C-thing part.
4. An X-thing is internally structured, so that any part of kind P1 is related to any part of kind P2 by an R-relation.
5. The environment of an X-thing includes an E-thing.
6. An X-thing is related by an S-relation to any E-thing in its environment.
7. An X-thing has the unary property Q.

A consistent collection of correct statements defines a model through the three basic classes of the ontological framework in Fig. 1. However, these classes are not suitable for storage of design information, since all connections between objects must be established. The trouble with this is that some connections might imply statements that the designer would not intend. In particular, some things are used in numerous different environments, and it would not be desirable to have to include all these environments in the definitions of thing classes like 'bolt' or 'nut'.

Instead, we have found that all information necessary for the creation of such a schema is contained in a consistent set of design statements, as related

above. Thus, the schema in **Fig. 2** has been developed to provide the data structure to be implemented. Design data are stored in instances of *ThingClassDefinition*, which collects instances from the six subclasses of *Attribute* that correspond to the six different kinds of attributes, see **Fig. 2**. The classes *RelationDefinition* and *UnaryAttributeDefinition* define relations and unary attributes, and ensures that attributes referred to in several places are identically defined.

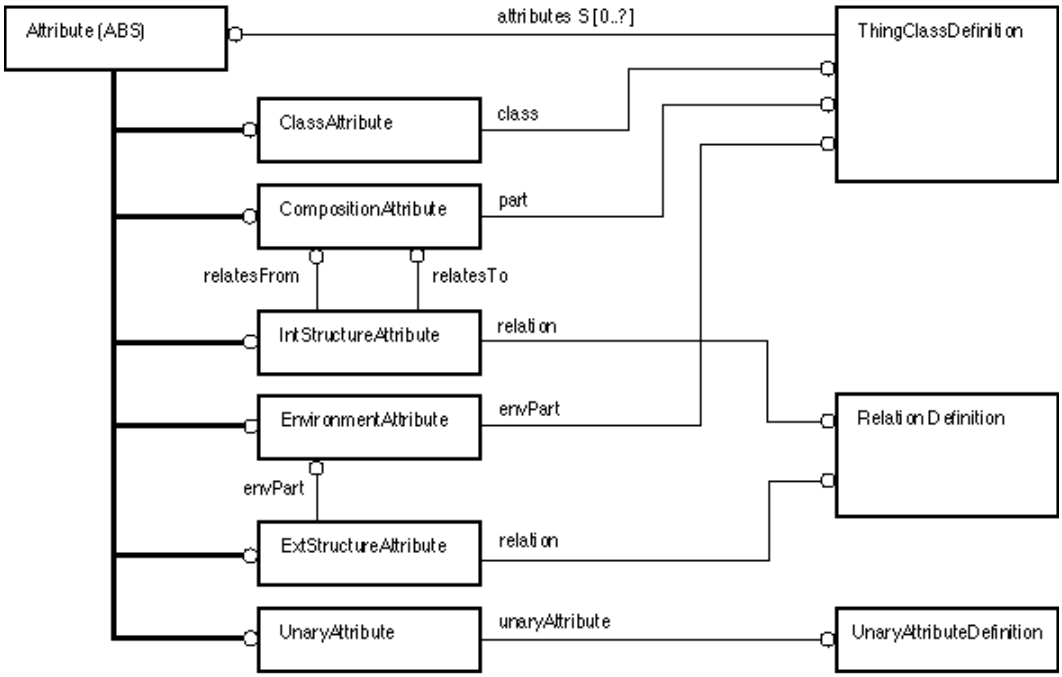

Fig. 2: Object structure of the current implementation

2.2 Example

To describe a thing in the BAS•CAAD system is to define thing classes for the thing and its parts, and the attributes that characterise those things. Fig. 3 displays a brief example of how an ordinary table may be described. The presentation is not an actual printout from the prototype software, although a similar syntax is used for files.

THINGCLASS *table*
  COMPOSITIONATTRIBUTE *tableTop* CARDINALITY *1*
  COMPOSITIONATTRIBUTE *tableLeg* CARDINALITY *4*
  INTERNALSTRUCTUREATTRIBUTE *tableLeg isConnectedTo tableTop*
  INTERNALSTRUCTUREATTRIBUTE *tableTop isConnectedTo tableLeg*

THINGCLASS *tableLeg*
  CLASSATTRIBUTE *woodenBar*
  ENVIRONMENTATTRIBUTE *tableTop*
  EXTERNALSTRUCTUREATTRIBUTE *isConnectedTo tableTop* IN *table*

THINGCLASS *woodenBar*
  UNARYATTRIBUTE *material* VALUESPACE *wood*
  UNARYATTRIBUTE *shape* VALUESPACE *bar*

Fig. 3: Thing class definitions

Table legs obviously are functional parts of tables; a wooden bar is characterised by its material, and its shape.

	House	Palce
Define attributes	man-made provides dwelling	Kind of House
Inherent attributes		man made provide dwelling

Fig. 4: Defined and inherited attributes

2.3 Other features of the object structure

Besides generating and maintaining the object structure, the prototype currently supports a simple schema for object identification, inheritance and specialisation of attributes, and class libraries. Additionally, class subsumption is planned to be implemented.

2.3.1 Object identification

The BAS•CAAD information system for design allows thing classes to be defined by references to predefined libraries (see section 3.3). Such a mechanism requires that libraries and classes can be unambiguously identified, in order to ensure that the correct libraries are used. The current version can identify objects within a library, but there is no secure identification of libraries. The development of such a mechanism is a task for database specialists and international standardisation organisations, and outside the scope of this phase of the BAS•CAAD research project.

2.3.2 Attribute inheritance

Class attributes indicate a more generic kind, also called a superclass. All statements defined for the superclass are also valid for the subclass; the attributes of the superclass are inherited attributes of the subclass. An example: the class House has the attributes ‘man-made’ and ‘provides dwelling’. If Palace is defined by the attribute ‘kind of Building’, Palace inherits the attributes ‘man-made’ and ‘provides dwelling’ (Fig. 4). A mechanism for attribute inheritance has been implemented.

2.3.3 Attribute specialisation

With the above set of attributes, however, the Palace class doesn’t reflect that a palace is a specific kind of house. This type of difference can be defined in to ways; either by adding a new attribute to the specific class, or by specialising an inherited attribute. Which way to choose is a question of clarity in modelling and usefulness of the model; se Fig. 5 for a illustration of the differences of the two methods in terms of sets.

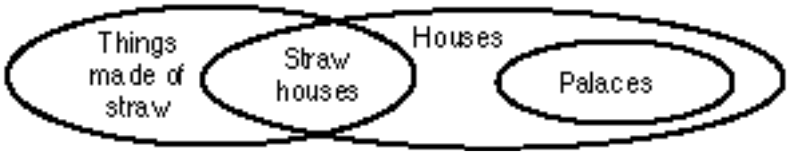

Fig. 5: Classes and subclasses

	House	Palace
Defined attributes	man-made provides dwelling	kind of House *provides luxurious dwelling*
Inherited attributes		man-made provides dwelling

Fig. 6: Attribute specialisation

In this case, we have chosen to specialise the attribute 'provides dwelling' into the more specific 'provides luxurious dwelling', and to define the class Palace with the specialised attribute. A mechanism for this has been implemented. The new attribute is defined in the Palace class, where it substitutes the inherited attribute 'provides dwelling'(Fig. 6).

2.3.4 Class subsumption

Description logics (see next paragraph) provide a mechanism called subsumption. This could be described as the inverse of inheritance, since it answers what superclasses a given class has according to its set of attributes. For instance, if a thing Bungalow has the attributes 'man-made' and 'provides dwelling', then it is subsumed to be a kind of House, i.e. the class Bungalow is a subclass of the class House. Subsumption is not currently implemented in the BAS•CAAD prototype system, since it would make implementing inference more complicated. However, since subsumption could arguably provide a powerful tool for case retrieval, it will probably be implemented in a future version of the BAS•CAAD system.

Description logics (DL) is a field within artificial intelligence research. It aims to develop mechanisms for describing concepts, and to automatically classify concepts. DL is based on first order predicate logic, and is DL systems implementations are usually similar to programming languages (Lambrix, 1996). Although DL is similar to the BAS•CAAD approach in many ways, it seems that it cannot serve as the base for a design information system. This subject is outside the scope of this paper, but will be further developed in a future publication by the present author.

3 The BAS•CAAD prototype information system

The prototype is intended to study the feasibility of organising an information system for design based on systems theory. It should also be dynamic in respect to the possibility for the user to define new class concepts in the conceptual schema, and to classify model instances.

The current implementation has been preceded by many earlier attempts. The foundation has in all cases been the concepts *system* and *property.* The aim of the BAS•CAAD project is to support expression of designed objects through these concepts in a computerised database. Thus, all versions of the software have sought to implement these two terms as object classes.

It is intended to display some features we consider important in an information system for design, but it is not intended for productive design work. The current version is centred on building symbolic schemas for concepts referring to things, and lacks most of the abilities to specify metric values necessary for a production tool.

3.1 Smalltalk

The BAS•CAAD prototype information system is currently implemented in Smalltalk under the Macintosh operating system. The reason for choosing the Smalltalk computer language was that it is object oriented, and that it supports explorative program development.

In product modelling, products are structured as objects that are assembled of objects in several levels. This makes it natural to choose an object oriented programming language for implementing such a system. Object oriented programming is based on the concept of objects that interact through sending and reacting to messages. Objects are defined through classes, which define the structure and behaviour of the object. The behaviour is the collection of various responses an object is capable of, and it is defined through pieces of software code, that in Smalltalk are called methods. Each kind of message that is 'understood' by an object corresponds to one method, which handles that particular kind of message. Some methods return answers, others just change the object's internal state.

In explorative program development, the software needs only to be partly defined before it is executed and tested. Thus, experimental solutions can be tried and kept if successful; otherwise, they are discarded. This way the final software solution is obtained though an exploration. In Smalltalk, code segments can be run and tried directly, without any time-consuming compilation or linking procedures. Actually, Smalltalk allows the programmer to change the code while the software is running, thereby relieving the programmer from repeatedly punching in lots of test data. This is not possible in C++, a popular programming language that also supports object oriented programming.

Smalltalk is profoundly object oriented; every concept is treated as an object class. Thus, in Smalltalk it requires an effort to not be object oriented, as opposed to C++. A drawback with Smalltalk is that the resulting programs do not run as fast as those created with e.g. C++.

3.2 Implementing the system

The foundation for the BAS•CAAD prototype is the three concepts thing class, relation and unary attribute that are implemented through the corresponding object classes *ThingClassDefinition*, *RelationDefinition* and *UnaryAttributeDefinition*.

To create and manipulate these objects, some means has had to be created. Principally, there are two choices: a command line user interface or a graphical user interface (a GUI). In a command line user interface, the user types commands, and then usually hits a key to get the command executed. A command line interface requires developing a command line interpreter, a piece of software that can translate the command lines into software actions and input data.

A GUI can make it easier for the software developer by controlling the user's actions. The user controls the software via buttons, menus, text input fields etc. Thus, the software developer can restrict the user to input only such instructions or data that are valid, through presenting only such control objects that allow actions that are meaningful to the software in a given situation. A benefit for the user is that it is fairly easy for the developer to make the software self-explanatory through a GUI.

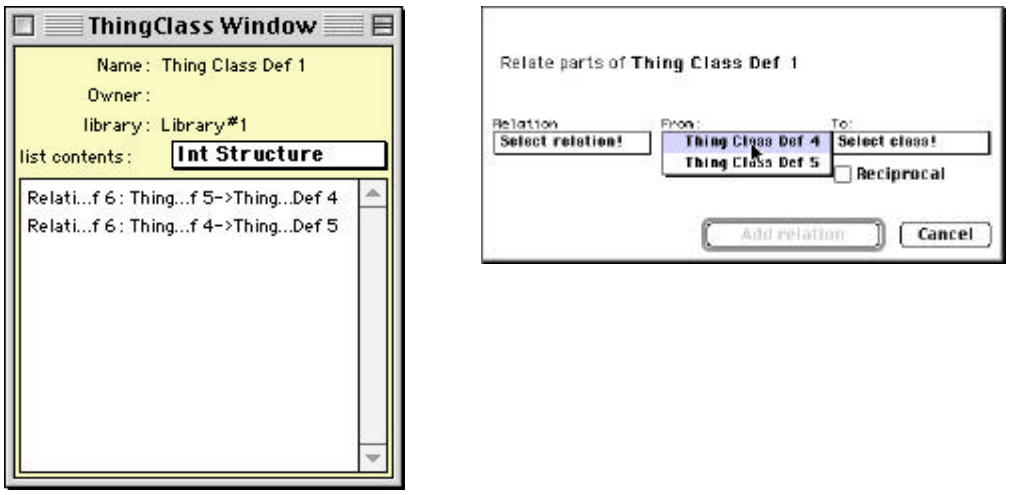

Fig. 7: Examples of user interface windows

Regarding these concerns, the choice fell upon a GUI as the method to interact with the software. Each of the object classes ThingClassDefinition, RelationDefinition and UnaryAttributeDefinition has a corresponding window class. The Smalltalk dialect used for the BAS•CAAD prototype (Smalltalk Agents, STA), provides generic classes of GUI components, such as windows, menus, buttons etc. These can be subclassed and modified to suit the particular needs of the software being developed.

In addition to the three classes of windows related above, which display the state of the model, additional windows have been developed for specific tasks, such as adding different kinds of attributes to thing class definitions.

Windows communicate with the corresponding model objects through messages. Each user interaction with a window causes at least one of the window's methods to be executed. This results in messages being sent to the model object, which responds with the appropriate actions. Finally, some messages are sent from the model object back to the window, to ensure that the window reflects the new state of the model object.

Most of the work with the development of this prototype system has been, and continues to be, to decide how to obtain the desired functionality. To perceive a user action in the terms of objects exchanging messages is a complex task. One particularly difficult question is to decide what object is the agent, and what objects are acted upon. The reason for this is that many objects may be involved in the execution of one user action, with many messages exchanged. If the resulting web of such message interchanges is too entangled, the programmer is likely to get lost. Additionally, there is a need to make the software code as intuitive as possible to any future programmer (which, incidentally, often is the same person, but a couple of days later).

3.3 Libraries

The use for the BAS•CAAD system is to record information about concrete things through the three concepts thing class, relation and unary attribute. The task for the user is to define what different kinds of things that compose the part of reality that is to be modelled, and the relations and other attributes that

characterise such kinds of things. Then, the BAS•CAAD system can be used to record these definitions.

Acknowledgeably, defining a coherent conceptual understanding of the concrete world is a very demanding task. To allow the BAS•CAAD system to be used in normal design situations, pre-defined well-considered schemas of thing classes has to be provided for use as libraries, from where the designer can fetch definitions to build up his design database.

The current BAS•CAAD prototype supports such class libraries. Objects in libraries can refer to objects in other libraries, so that hierarchies of libraries can be created. This feature is implemented to try a mechanism that would allow international and national standardisation organisations to create generic libraries. Building material providers and others can then create intermediate level libraries that ultimately can be used as references in specific building projects.

The concept of class library has been generalised, so that library is the sole format for databases and files. Thus, the work of any level of specificity can be the foundation for further specialisation. For instance, it would be possible to use the project file from a window designer as a library file in a building design project without any reformatting or reclassification.

4 References

Eastman C. M. and Fereshetian N. (1994). Information models for use in product design: a comparison. *Computer-Aided Design* Vol. 26, No 7, pp 551-572, July 1994.

Eastman C. M., Assal H., and Jeng T. (1995). *Structure of a database supporting model evolution*. In *Modelling of buildings through their life-cycle.* Proceedings of CIB workshop on computers and information in construction (eds. Fisher M., Law K., and Luiten B.) Stanford University, Stanford, Ca, USA, August 21-23.

Eastman C. M. and Siabiris A. (1995). *A generic building product model incorporating building type information*. Automation in Construction, vol. 3, no. 4, pp. 283-304.

Ekholm, A. and Fridqvist, S. (1998) *A Dynamic Information System for Design Applied to the Construction Context.* In *The Life-cycle of Construction IT Innovations* (Eds. Björk, B-C. and Jägbeck, A.), proceedings from the CIB W78 workshop, 3-5 June 1998, Stockholm, Sweden

Fridqvist S. and Ekholm A. (1996). *Basic ObjectStructure for Computer Aided Modelling in Building Design*. In *Construction on the Information Highway*. (Ed. Ziga Turk), proceedings from the CIB W78 Workshop, 10-12 June 1996, Bled, Slovenia.

Galle P. (1995). *Towards integrated, "intelligent", and compliant computer modeling of buildings*. Automation in Construction. Vol. 4, No 3, pp. 189-211, 1995.

Junge R., Steinmann R. and Beetz K. (1997) *A dynamic product model*. In *CAAD futures 1997,* proceedings of the 7th International Conference on Computer Aided Architectural Design Futures (Ed. Richard Junge) Dordrecht: Kluwer Academic Publishers.

Lambrix P. (1996) *Part-Wole Reasoning in Description Logics*, dissertation at the Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Leeuwen J. P., and Wagter H. (1998). *A Features Framework for Architectural Information*. In the proceedings of the Artificial Intelligence in Design Conference 1998 (Ed. Gero, J. And Sudweeks, F.) Dordrecht: Kluwer Academic Publishers.

Maher M. L., Simoff S. J. and Mitchell J. (1997). *Formalising building requirements using an Activity/Space Model*. Automation in Construction, vol. 6, pp. 77-95.

Schön D. (1983) *The reflective practitioner*. BasicBooks 1983.