A Software Architecture for Self-updating Life-cycle Building Models

BRUNNER Klaus A. and MAHDAVI Ardeshir
Dept. of Building Physics and Building Ecology, Vienna University of Technology, Austria

Keywords: building models, sensors, simulation-based control, life-cycle

Abstract: This paper describes a computational infrastructure for the realization of a self-updating building information model conceived in the design phase and carried over to the operation phase of a building. As such, it illustrates how computational representations of buildings, which typically serve design support, documentation, and communication functions, can be transported into the post-construction phase. Toward this end, we formulate a number of requirements for the conception and maintenance of life-cycle building information models. Moreover, we describe the architecture and the prototypical implementation of such a model.

1 INTRODUCTION

Computational representations of buildings typically serve design support, documentation, and communication functions. A seamless transition from such a representation into the post-construction phase of a building's life-cycle is, however, highly desirable: Such a transition would eliminate – or at least reduce – the redundancy involved in the generation of a high-resolution building model at the outset of the operational phase of a building. However, precise requirements for the conception and maintenance of life-cycle building information models are yet to be formulated. Effective life-cycle models need to be comprehensive, detailed, multi-aspect, and self-updating. They must provide for scalable dispositions to receive and process dynamic (sensor-based) messages regarding changes in buildings' configuration and status. They must also accommodate the informational requirements of operational applications for building control and management (Mahdavi 2004a).

To address these issues, we describe a computational infrastructure for the realization of a self-updating building information model conceived in the design phase and carried over to the operation phase of the building.

2 PROJECT DESCRIPTION

In the following sections we discuss the main requirements of a building model service and the approaches taken in our prototype's design to meet them.

2.1 Requirements

From the outset, the three main tasks in the operation of a building model service – or any model service, for that matter – can be summarized as follows: 1) data import, 2) model representation and operation, and 3) data retrieval.

2.1.1 Data Import

In the operational phase of the building, its model must be kept up-to-date with current data, preferably collected automatically by sensors placed throughout the building. These may include occupancy sensors, temperature and humidity sensors, inventory tracking sensors, and sensors built into various technical subsystems such as HVAC, shading, and lighting systems to report their current status. The requirements are thus:

Low latency. Sensor readings should be conveyed to the model as quickly as possible. Some control systems (e.g. for lighting) must react within seconds to changes in the building or its environment, requiring an up-to-date model for their control decisions.

Scalability. A large number of sensors – conceivably thousands in a large office building – must be supported.

Adaptability. Sensors use a wide variety of interfaces and protocols to report their readings. It must be possible to convert and import sensor data from these devices with minimal effort.

Low maintenance. Adding or removing sensors should be as simple as possible, with minimal operator effort. This can also be seen as a measure of robustness: a few failing sensors should not disrupt the entire system's operation.

2.1.2 Model Operation

The model service must be able to maintain a consistent state of all available building information and store it efficiently, for access to current and historic data. The main requirements are:

Rich object model. The building object model should be expressive enough to store geometric and semantic information for a wide range of applications.

Concurrency management. The model must be able to maintain a consistent state while allowing, ideally, concurrent read and write access for multiple processes.

Storage management. The service should not just keep current state and allow efficient queries for various attributes, but also store historic model data for later retrieval in an efficient manner.

2.1.3 Data Retrieval

A wide range of client applications may be interested in model data, from end-user visualization to heating control applications. Some need a snapshot of the model in its entirety, others may require selected spatial or temporal portions of the model. The main requirements are thus:

Scalability. Concurrent access by multiple client applications must be possible without disrupting the model service's performance.

Versatility. While some applications may require a simple "snapshot" of the model, others will be interested in specific events (e.g. a sudden drop in temperature, or a change in occupancy of a given space). In some cases, a history of events in a given space can be useful, e.g. to analyze the performance of heating control over the course of a day. The model service must be able to cater for these very different client needs and deliver data in the necessary formats.

2.2 Architecture

In the following subsections we describe the main elements of our prototype's design, based on the tasks outlined in the previous section.

2.2.1 Related Work

Recently, a growing number of publications has been published in research on *sensor networks*. Sensors are envisioned as smart, networked devices embedded into the spaces and objects they are monitoring (Akyildiz 2002). Architectures for transporting sensor readings and querying the sensor network have been proposed (Bonnet, Gehrke, and Seshradi 2001). Sensor networks could serve as a source of building-related data that have been hard to obtain in an automated manner until now.

The work on building product modeling has a relatively long tradition. Specifically, there have been numerous efforts to arrive at systematic and interoperable computational representations for building elements, components, and systems (IAI 2004, ISO 2003, Mahdavi 2004a). However, only recently the concept of real-time sensor-supported self-updating models toward supporting building operation, control, and management activities has emerged (Mahdavi 2004b). In the present contribution, we specifically focus on the integration of sensory networks, building models, and model-based process control applications.

2.2.2 Data Import

To facilitate a simple but powerful communication system between data producers (sensors), the model service, and client applications, a solution based on tuple spaces was chosen. Space-based systems (Carriero, and Gelernter 1989) can accommodate a wide range of interaction patterns in distributed systems, while making the actual details of distribution almost completely transparent.

The JavaSpaces specification (Freeman, Hupfer, and Arnold 1999) describes such a system for the Java programming language. Processes communicate through a simple interface offering two main operations (*write object* and *take object*) on a distributed, logically shared data structure with the characteristics of an associative memory (“bag”). Applications may take objects from the space based on their type and the values of public fields, both synchronously (waiting until an appropriate object appears) and asynchronously (application is notified by means of a call-back mechanism when an object becomes available). As Noble and Zlateva (2001) demonstrate, JavaSpaces are not the fastest way of sending objects between two hosts in a network. The benefits of JavaSpaces lie in their simplicity and the uncoupling of the communicating programs in time and space, taking advantage of which requires a different design approach than other distributed programming methodologies.

One typical application is workload distribution: clients can post work requests into the space, while any number of worker processes can take these requests from the space, process them, and post the results back to the space for clients to take. The remarkable aspects here are the transparency of distribution and the low level of coupling among all the parties: neither clients nor workers need to know anything about each other except the signature of the work request and result objects, and there is no requirement for synchronous operation (as in a normal remote procedure call). Clients may choose to submit a work request, wait for the response, and post the next request; or they may post a batch of work requests at once and come back later to pick up the results. As long as clients and workers are only interested in single objects, concurrency issues are handled transparently by the space.

These characteristics match the requirements for data import very well. Data producers (sensors) can post sensor readings to a space without regard for other data producer and how (or if) the data are actually processed. The space acts as both a buffer and a transport mechanism to one or more data consumers, such as the model service.

At this low level – which deals only with transporting streams of sensor readings, not with semantics and the model context – there is no need to specifically register or de-register new sensors: data producers can appear (start putting objects to the space) and disappear (stop putting objects to the space) any time. Once an object is taken from the space, it is not available to other consumers any more. As there may be multiple unrelated consumers interested in sensor readings, a subscriptions system is used. Groups of equivalent consumers (comprising any number of consumers) can register their interest in sensor readings by writing to a singleton registration object in the space. For each subscribed consumer group, one sensor readings object is written to the space by the data producers (see Figure 1). This evidently means that *n* copies of each object must be written for *n* consumer groups, but saves multiple read and write operations to shared synchronization objects for both senders and receivers as required by other space communication patterns, such as “Channel” (Freeman, Hupfer, and Arnold 1999). The number of consumer groups is assumed to be very small even in large setups; in our prototype, there are only two groups: the model service, and an additional data broker service keeping track of sensor readings mainly for debugging purposes.

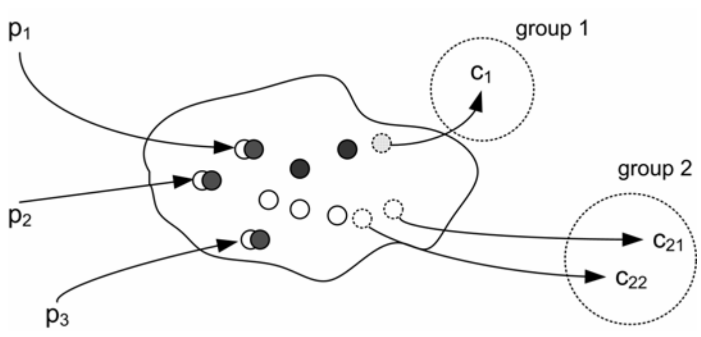

Figure 1 Sensor Data Import. Three consumers in two consumer groups

The space’s buffering characteristics implicitly improve the reliability of the system: if one data consumer is offline (e.g. for maintenance, or due to a software crash), data will either be made processed by another consumer from its group if one exists, or it will be buffered transparently for a certain time to be processed as soon as a consumer is online again.

Our current approach to importing data is a *push* model: sensors post information according to their own schedule; data consumers may decide to pick these data up or ignore them. However, the architecture does not preclude a *pull* model, as it is also possible to have sensor adapters listen for requests and post readings in response.

2.2.3 Model Representation and Operation

Model Representation

Our building model is based on the Shared Object Model (SOM), which has been designed to allow the automated derivation of domain-specific models for various applications (Mahdavi, Suter, and Ries 2002; see also Figure 2). In the current implementation, the entire building model’s current state is kept in main memory.

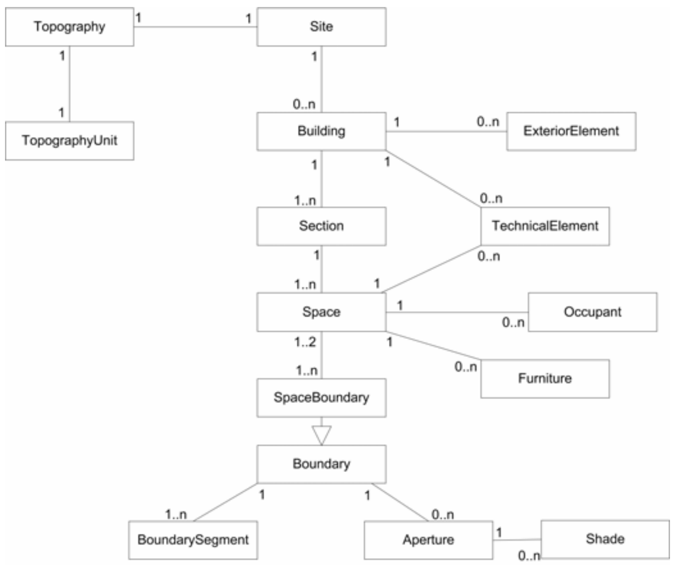

Figure 2 Shared Object Model – SOM (simplified)

Concurrency, Persistent Storage and Retrieval

Each SOM object is tagged with an ID and a ticket obtained from a global, monotonously increasing counter. As soon as an object changes, a new ticket is obtained, and a copy of the object's data is queued for background serialization to the on-disk database. Additionally, a series of updates on more than one object (e.g. when a moving object has to be disassociated from one space and associated to another) can be aggregated into one atomic change operation, in order to prevent inconsistent tree states from becoming visible to other readers. A mapping of tickets to actual time is kept for later queries.

If the state of an object at a given clock time t is requested, first the corresponding ticket value v is retrieved (if there are multiple ticket values for a given time, the highest value will be chosen). The database will then be queried for the object with the given id and the highest ticket value that is less than or equal v. If parent or child objects (e.g. the boundaries of a space) of the retrieved object are requested, the procedure is the same.

This storage and retrieval system is based on the overlapping trees method (Burton et al. 1985) and multi-version database systems. Multi-version databases, as shown by Buckley and Silberschatz (1983), have the potential for drastically reducing concurrency delays caused by locking. The design ensures that a consistent state of the entire object tree can be restored for any given point in transaction time, while keeping storage use low by only storing changed objects.

Sensor Discovery

One of the requirements described under data import is low maintenance. As shown above, new sensors do not have to be registered explicitly to be able to send data. However, the model service must be able to put these new sensor readings in context: e.g., a source of temperature data must be related to a location to become a meaningful part of the model and to allow queries such as “what is the current temperature in space *x*?”. While it is possible for an operator to enter this relation manually, the model service can also use location information for the given sensor obtained from a location-sensing system (Icoglu et al. 2004). Upon encountering a new source of sensor data, the model service will register its identification and wait until the respective location information for it is received. Once this has occurred, a sensor object is instantiated and linked to the space containing the given coordinates.

2.2.4 Data Retrieval

Simple procedure-call interfaces are not powerful enough to allow for the range of queries that client applications might require. The common alternative approach is to devise a rich query language to express what kind of information is requested from the model. The client thus has to translate its request into the query language, which is in turn parsed by the server and used to select the appropriate objects from the data set and send these to the client. This approach has been used successfully for years in the world of relational databases, namely, as Structured Query Language (SQL). Besides its advantages (e.g. the simple text-based input and output, which eases network transport and integration with all kinds of different client implementations), the intermediate steps of translating query language and converting the server's response to the format required by the client are time-consuming and cumbersome. Additionally, as a pure query language, it does not have the semantic richness of a turing-complete programming language. Complex tasks may have to be performed as a series of queries, increasing the amount of communication necessary between client and server.

In our building model service, we take the alternative approach of allowing clients to submit *agents* to the model service. Agents are submitted as Java objects through the service space and started as separate threads by the model service. They can traverse the model and inspect its objects through a Visitor-style interface. They may also register to be notified of changes in parts of the model: e.g., a lighting control application can submit an agent that monitors a given workplace. It may inspect the relevant SOM objects and derive a suitable representation of the space for a lighting simulation application, whose results in turn will be used by the lighting controller to take the necessary actions. The agent may then register its interest in changes and put its executing thread in “sleep” mode. When as a significant change in the model occurs (e.g. the readings of an illuminance sensor drop sharply, or changes in the workplace geometry are reported by location sensors), the agent is notified by the model service and can now re-inspect the model and inform the control application if necessary. In the terminology of Nwana (1996), our agents are static and not collaborative, although collaboration may be investigated for future applications.

This approach allows complex interactions with the model and reduces network traffic by bringing code and the data it is operating on close together during execution. Developers of client applications can develop their software directly in terms of objects of an expressive model structure, without having to translate into an intermediate language. The service space facilitates loosely coupled communication between system components: tasks that do not involve direct transactions with the model, or that have to provide a user interface, can still be separated into different processes, possibly on different machines.

2.3 Initial Implementation

Figure 3 shows our experimental implementation. To the left, the *data space* as described earlier is shown. Its main function is to convey sensor information to the model service, but also to send commands to the actuators. In our case, a lighting and shading control system (Luxmate®) is being used to control light fixtures and external shading; indoor sensors for illuminance, temperature, occupancy etc. and outdoor weather station sensors are polled using LabVIEW. Both systems are connected to the data space with specific adapter programs written in Java. All space communication is using standard TCP/IP infrastructure.

To the right, the *service space* mediates requests and responses between the various services and client applications, most prominently the building model service and a lighting controller application. Lumina, an interior lighting simulation application (Pal and Mahdavi 1999) written in C++ and connected to the space through a simple adapter program, is used by the controller to assess the effects of various control actions on the space (e.g. dimming lights, adjusting shades), taking into account the geometry of the space, the states of light fixtures and shades, and exterior lighting conditions. All these data are acquired and converted to Lumina's input format by a software agent submitted to the model service by the controller.

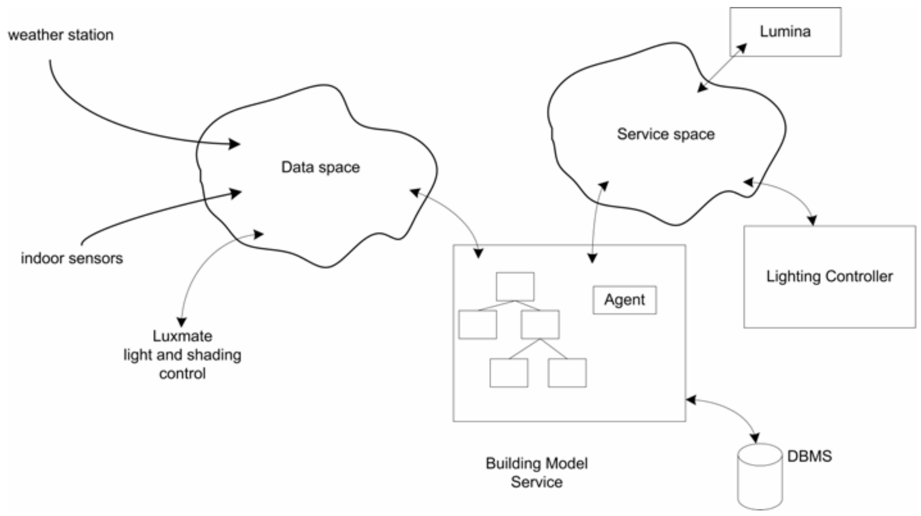

Figure 3 Overview of Experimental Setup. Arrows denote data flow

Our preliminary tests were conducted in an office space containing two desks, four light fixtures and motorized shading for two windows, supplemented with data from an external weather station. A commercial JavaSpaces implementation (GigaSpaces) was used for the data and service spaces. Initial results show that the system is capable of handling concurrent input from all data sources over the course of months without interruption, storing about 4 Gigabytes of sensor-supplied data. Latency (the interval between the instant of reading a measurement from the sensor by LabVIEW, and the instant it becomes available in the model service) has been consistently less than 0.5 seconds. Model operation and data export have been found to work correctly and reliably.

3 CONCLUSION

We have shown requirements for an architecture for self-updating life-cycle building models and a prototype implementation addressing these requirements. A communication system for sensor readings, actuator commands, and client/server communication based on JavaSpaces has been tested successfully. A model service design based on SOM and software agents was shown to be feasible. We believe that the presented approach would allow for the consideration of complex issues pertaining to the operation and maintenance of sophisticated building service systems already in the design phase of buildings.

There are a number of areas for further research. One of the next tasks will be choosing an appropriate spatial indexing mechanism for large models. Building models contain a large number of mostly static objects that could serve as natural partitioning boundaries for spatial indexing, but also a number of moving objects that tend to move along "paths" that emerge over time. Most spatial indexing schemes, even those optimized towards moving objects, are not suited for this kind of data (Gaede and Günther 1998). Moreover, the current storage and retrieval system's performance should be measured under realistic conditions. The choice of granularity for database storage affects its space- and time-efficiency and should therefore be based on actual usage patterns.

ACKNOWLEDGEMENTS

The research presented in this paper has been supported by a grant from FWF (Austrian Science Foundation), project number P15998-N07.

REFERENCES

Akyildiz, I. F., W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. Wireless sensor networks: a survey. *Computer Networks* 38(4): 393–422.

Bonnet, P., J. Gehrke, and P. Seshadri. 2001. Towards sensor database systems. In *MDM '01: Proceedings of the Second International Conference on Mobile Data Management*, Number 1987 in Lecture Notes in Computer Science, 3–14. Berlin: Springer-Verlag.

Buckley, G.N., and A. Silberschatz. 1983. Obtaining progressive protocols for a simple multiversion database model. In *Proceedings of the 9th Int. Conference on Very Large Data Bases*, 74–80. San Francisco: Morgan Kaufmann.

Burton, F.W., J.G. Kollias, D.G. Matsakis, and V.G. Kollias. 1990. Implementation of overlapping B-trees for time and space efficient representation of collections of similar files. *Computer Journal* 33(3): 279–280.

Carriero, N., and D. Gelernter. 1989. Linda in context. *Comm. ACM* 32(4): 444–458.

Freeman, E., S. Hupfer, and K. Arnold. 1999. *Javaspaces Principles, Patterns, and Practice*. Boston: Addision-Wesley.

Gaede, V., and O. Günther. 1998. Multidimensional access methods. *ACM Comput. Surv.* 30(2): 170–231.

International Alliance for Interoperability. 2004. Industry Foundation Classes. Available from http://www.iai-international.org/iai_international/Technical_Documents/iai_documents.html. Internet. Accessed 1 February 2005.

Icoglu, O., K. A. Brunner, A. Mahdavi, and G. Suter. 2004. A distributed location sensing platform for dynamic building models. In *Proceedings of the Second European Symposium on Ambient Intelligence*, Number 3295 in Lecture Notes in Computer Science, 124–135. Berlin: Springer-Verlag.

International Organization for Standardization. 2003. ISO-STEP part standards TC184/SC4.

Mahdavi, A. 2004a. A combined product-process model for building systems control. In *eWork and eBusiness in Architecture, Engineering and Construction: Proceedings of the 5th ECPPM Conference*, 127–134. Leiden: Balkema Publishers.

Mahdavi, A. 2004b. Reflections on computational building models. *Building and Environment* 39(8): 913–925.

Mahdavi, A., G. Suter, and R. Ries. 2002. A representation scheme for integrated building performance analysis. In *Proceedings of the 6th International Conference on Design and Decision Support Systems in Architecture*, ed. H. Timmermans, 301–316. Avegoor: The Netherlands.

Noble, M.S., and S. Zlateva. 2001. Scientific computation with Javaspaces. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, 657–666. Berlin: Springer-Verlag.

Nwana, H.S. 1996. Software agents: An overview. *Knowl. Eng. Rev.* 11(3) 1–40.

Pal, V., and A. Mahdavi. 1999. A comprehensive approach to modeling and evaluating the visual environment in buildings. In *Proceedings of Building Simulation 99 – Sixth International IBPSA Conference* [Volume 2], eds. N. Nakahara, H. Yoshida, M. Udagawa, and J. Hensen: 579–586. Kyoto: Japan.