The software design of a dynamic building model service

K. A. Brunner & A. Mahdavi
Department of Building Physics and Building Ecology, Vienna University of Technology, Austria

ABSTRACT: We present the software architecture and a prototypical implementation of a dynamic building model service system. The primary purpose of this model service is to support (simulation-assisted) indoor-environmental control operations in buildings. However, as a comprehensive, structured, sensor-based, and self-updating information resource, the model can support other building tasks such as those concerned with building logistics and management. At the core of our model service design, an object tree continuously updated from sensor data reflects the current state of the building, concurrently accessible to multiple clients and backed by persistent storage. The service is embedded in a distributed infrastructure based on tuple spaces for transparent object-based communication between system components. The preliminary evaluation of the model service system suggests that the proposed design is feasible and appropriate for further testing in realistic (large-scale) settings.

1 INTRODUCTION

This paper presents a software design for dynamic building model services. As such, it represents a component of ongoing work on a larger research project toward realization of sentient buildings (Mahdavi 2004). Sentient buildings possess an internal, dynamic, and self-updating representation (model) of themselves. They use this model to support various services and operations. The research project focuses on the application of such internal models toward supporting indoor-environmental control systems of buildings (e.g. heating, cooling, ventilation, and illumination systems). Specifically, we have been investigating the potential of dynamic building models to enable simulation-based building control strategies (Mahdavi 1997, 2001, Clarke et al. 2001). To identify a desirable state for a building control device, the simulation-based control method projects a number of alternative device states into the future, predicts the implications of these alternative states via simulation, compares the simulation results in view of pertinent objective functions, identifies the most preferable device state, and informs the user (or instructs a relevant actuator) toward the realization of this state. As compared with traditional control algorithms, simulation-based strategies have been shown to be highly effective in the context of built environment. This is primarily due to two circumstances: *i)* building control operation involves a large number of environmental sub-systems and a multitude of devices and networks; *ii)* buildings are subject to both dynamic contextual forces (e.g. weather conditions) and internal fluctuations (e.g. occupancy presence and actions) that are difficult to predict.

Whilst simulation-based control strategies have a number of advantages, they are not easy to implement. First, they require a fairly detailed model of the building, its systems, its context, and its occupancy. Second, given the dynamic nature of building-related processes, such a model must be continuously updated to be reliable. Advances in computer hardware and simulation algorithms have brought simulation times to a level that is useable for BEMS (Building Energy Management Systems) applications. However, creating simulation models is still, to varying extent, manual labour. The transition from initial CAD (computer-aided design) building documents to simulation models is hardly seamless and often requires additional domain-specific information and extensive post-processing. Moreover, any significant change in the building must also be reflected in the simulation models, if they are going to be of any use in the context of building systems control. Ideally, simulation-based control requires a model of the building's status that is updated without human intervention. This evidently requires an extensive sensor infrastructure in the building generating a huge amount of raw data – and consequently, software that processes these data, collating and organizing them contextually for access by other software. We believe that such a dynamic building model would be useful for many other purposes besides simulation-based building systems control, offering a level of abstraction and a common interface that has not been available so far.

Today, modern office buildings are often equipped with considerable networks of sensors and actuators. However, there is generally a lack of meaningful integration and open access to make full use of these available data. We therefore propose a dynamic building model service to address this need, outlining requirements and a prototype software design, and report our experiences with an actual implementation.

2 BACKGROUND

Considerable work has been done in the field of building product modelling (Eastman 1999, Mahdavi et al. 1999). Product model specifications formally describe structures and notations and thus serve as a conceptual basis for building models; however, they do not address the architecture and run-time behaviour of a building model service.

Work on communications infrastructure for sensors and actuators within buildings has resulted in many specifications and products, e.g. BACnet, LonWorks, LUXMATE and others (Bushby 1997, Sharples et al. 1999, Luxmate 2005). A building model service naturally relies on some form of communications infrastructure and should be easily adaptable to specific variants.

Recently, the integration of various control domains has become a focus of research in building energy management systems (BEMS). The EDIFICIO project (Guillemin & Morel 2001) has shown the use of soft computing techniques applied to concurrent control of heating, ventilation, and lighting. Simulation-based control has been argued for and demonstrated successfully by Mahdavi (2001) and Clarke et al. (2001). The models used in these instances were specialised to the given experimental setups and control tasks and not designed to be scalable or usable for multiple applications concurrently.

The S2 project (Mahdavi et al. 1999) demonstrated automated derivation of domain-specific models for simulation from a general building model, in a distributed environment. However, it was geared toward the design phase only and did not support simulation-based control or dynamic building model updating during its operational phase.

3 DESIGN AND IMPLEMENTATION

In this section, we outline the design of the model service and discuss its key elements. Functional and non-functional requirements for a dynamic building model service are stated in section 3.1 and previous work (Brunner & Mahdavi 2005). The most important non-functional requirements are *scalability* and *versatility*. Scalability relates to the need to handle large buildings with great numbers of spaces, sensors, actuators, and other elements efficiently. Versatility (or flexibility) means that the model service must be able to accommodate a wide range of different uses and application software: this suggests a lean core application that can be extended during run-time with additional data and behaviours as needed.

3.1 *Concepts*

An *integrated* building model comprises information on all elements of a building to a level of detail that is sufficient to support a wide range of applications, such as photorealistic rendering, occupancy monitoring, and thermal simulation. Contrary to domain-specific, parametric models such as those used in model-based control (Pargfrieder and Jörgl 2002), it must be designed to hold multi-aspect, multi-purpose data and to be openly accessible for any application through a well-defined interface.

Unlike a simple database of raw values, data are organized in the form of a well-defined *object-oriented product model* providing context and semantics.

A *dynamic building model service* is updated regularly, e.g. through sensor readings, to reflect the current state of the building as accurately as possible at all times. As it does not merely store the received data, but can also apply some processing to them or reconfigure itself if necessary, it can be seen as "self-updating". Thus, the model is not merely a description used for reference and off-line analysis; it is a live object tree to be used by any number of applications during the building's operation, processing data updates and application requests concurrently. Input data may come from a range of different sources and must be correctly placed in model context. Such data updates can happen continuously (e.g. a stream of measurement values from an illuminance sensor) and must be processed within a short timeframe to meet the requirements of control applications. Additionally, it is desirable that not just the current, but all historic states of the model are stored persistently to be easily retrieved.

3.2 *Centralised vs. decentralised architecture*

Some types of applications (e.g. thermal simulation) span the entire building, while others (e.g. lighting control for a windowless room) may be focused on just a small portion. This determines their usage patterns for building data and suggests different designs: an all-encompassing central model service allowing random access to any portion of the building, or multiple, loosely connected or even independent sub-model services focused on different parts of the building (Sharples et al. 1999).

The burden of model-keeping for a centralised model in terms of memory and CPU usage may be huge. A central model has to receive data from all sources in the building, essentially forming a bottleneck in the data flow. Decentralised model services could be distributed to different computers for workload distribution and shorter data paths.

Decentralisation means that it must be decided from the outset how the entire model is broken into parts. However, it is hard to find an optimal division scheme for all possible applications. While many applications lend themselves easily to a division along units such as “floor”, “apartment”, “room”, some building systems work across these lines, such as elevators or HVAC piping. Applications monitoring these systems would have to be in constant communication with multiple sub-models, increasing network traffic and CPU loads.

As versatility is a key requirement of our specification, we have opted for a centralised model design. The full current state of the model is kept in working memory of a single process on one computer. However, it is possible to extract copies of parts or the entire object tree for off-line analysis. This way, an application working repeatedly on a relatively static portion of the model can be fully decoupled from the model service.

The Java language system was chosen as our implementation platform mainly for reasons of operating system independence, good availability of third-party class libraries and mature facilities for distributed computing.

3.3 *Interconnection of system components*

In our project, the model service is part of a distributed infrastructure that comprises a number of other services that depend on or assist the model’s operation (Brunner & Mahdavi 2005). As a design guideline, we identify two types of runtime behaviour in terms of model access patterns:

a) *Batch* behaviour: a module collects some input data, performs intensive processing on it, and returns some output data. One example is model-based lighting simulation (by ray tracing or radiosity), another is spatial reasoning (e.g. to generate space boundaries from tag locations). Modules of this kind are essentially services operating on a request-response basis.

b) *Interactive* behaviour: a module keeps accessing a number of objects repeatedly, possibly reacting to events and changing the objects. It requires little processing power, but low-latency object access. One example is a lighting controller task that monitors workplaces and registers any relevant events that may occur, e.g. changes in occupancy or daylight.

Modules with *batch* behaviour benefit from distribution to keep high CPU workloads off the computer hosting the model service. To achieve this distribution, we are using a tuple spaces system based on JavaSpaces (Freeman et al. 1999). For instance, a client’s request for lighting simulation can be posted to the *service* space and subsequently picked up and processed by any connected machine running an instance of such a service. Once completed, the results are placed back into the space to be picked up by the client. This allows a simple and transparent load distribution that decouples modules in time and space as much as desirable. Neither clients nor servers need to know anything about each other except how to access the common space and the signature of the relevant request and response objects. Clients can choose a synchronous or asynchronous mode of operation: either posting requests and waiting for responses sequentially, or posting a batch of requests at once and coming back later to pick up the results. The latter scenario is particularly suited for simulation-based control programs, which frequently need to commission a set of simulations to select the best control decision.

The characteristics of modules with *interactive* behaviour suggest a different approach. It is desirable to keep these modules’ code close to the data during runtime without losing the flexibility and loose coupling of the system by hard-wiring their code into the model service. One way to achieve this goal is to design an elaborate query language for the model: the main advantage of this approach is that modules are not bound to any specific programming language, as long as they can submit well-formed query strings to the model service and process the results. However, the developer effort of translating query or program logic into an intermediary is considerable, and there would be significant communications overhead incurred by repeated queries and responses. Ideally, modules should be able to access the model just like any other Java object.

This is achieved by implementing them as *agents*, which might also be called “mobile plug-ins”. Agents are thread objects that may be submitted to the model service over the JavaSpace, where they are started inside the service process. They can directly access the object tree and use all public operations on the objects as well as a number of utility methods for traversing the object tree, retrieving historic versions, and communicating with other modules via the JavaSpace. Moreover, agents can register for events on specific objects to be notified of data updates.

As an example, the core of a heating control application can be sent to the JavaSpace, where it can examine the relevant objects, derive a number of possible control decisions and send a batch of simulation requests (containing relevant model data) to the JavaSpace. Using the results provided by one or more simulation services connected to the JavaSpace, the controller can take appropriate action (e.g. opening a valve). Further control cycles can either be triggered in time intervals or based on update events (e.g. when a temperature sensor reading rises above a threshold).

For both batch and interactive access patterns, the proposed design ensures modularity and flexibility while the specific runtime characteristics are taken into account.

3.4 *SOM objects*

We have chosen the Shared Object Model SOM (Mahdavi et al. 1999) as basis for our model. SOM defines only a core set of general attributes (mainly geometry and surface properties, and a few others depending on the object type) for the various elements of a building, as it was not designed to be a model for all conceivable building applications. Domain-specific data (e.g. the photometric characteristics of a light fixture) can be associated as separate objects if necessary.

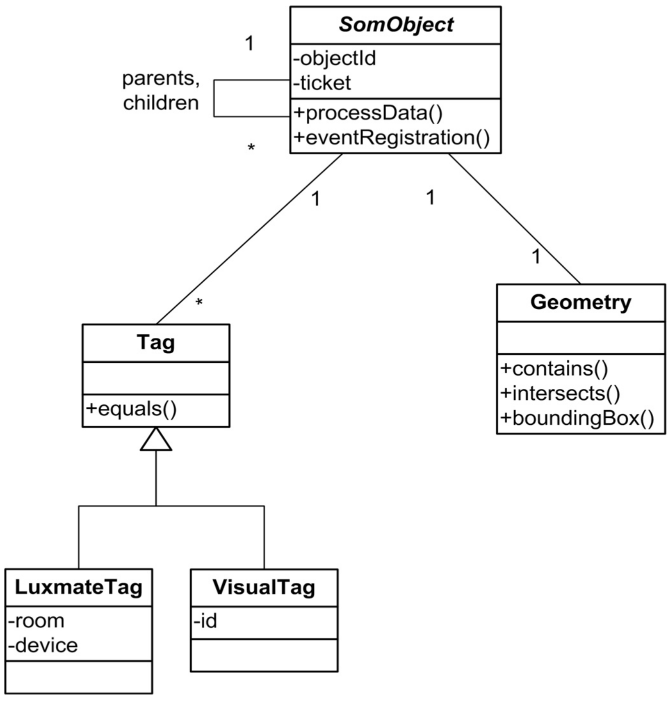

Figure 1. A generic SOM object and some of its most important associations, methods, and attributes (simplified).

Each SOM object can be associated at run-time with a number of *tags*. Tag objects represent keys for the given object: for instance, a light fixture's address in the LUXMATE device control network is one kind of tag; the ID of a physical location sensing tag attached to the same light fixture is another. In both cases, the tag object serves as a key for the model service to route incoming sensor data to the proper object. SOM objects may have child and parent objects (Figure 1).

Note that, while we use SOM as the underlying building model, the proposed software design does not preclude the use of other models (such as IFC). As our past research has shown, SOM classes can be effectively mapped to IFC classes, thus ensuring the interoperability of our developments with IFC-compliant applications (Lam et al. 2003).

3.5 *Sensor and actuator communications*

As input data may come from a variety of source at the same time, multiple threads can be working on reading data and routing it to the proper objects, based on their tags. For our experimental setup, we use a separate JavaSpace that holds data from different sources, such as illuminance sensors and an indoor location sensing system (Icoglu et al. 2004, Brunner & Mahdavi 2005). Distributed sensors push data objects to the space which are picked up by the model service.

The primary task of the input worker threads is to find the relevant SOM objects (based on their associated Tags) and call their *processData()* method. This method decides if and how incoming data is processed, which may result in updates on the object's data or the creation and deletion of new child objects. With the help of location sensing data, new objects can, to some extent, be automatically included in the building model: a temperature sensor that is marked with a location sensing tag will be represented by a new SOM object and attached to the proper space as soon as both location sensing and sensor reading data are available. We plan to reach a further degree of automation through spatial reasoning to automatically discover the boundaries of spaces (Suter et al. 2005).

To handle new sensor data types, data handler objects can be dynamically registered with the model service at runtime. These are used if a SOM object signals that it is unable to process the received data.

For actuators, SOM objects also serve as interfaces to initiate physical change. For instance, an agent can change the dimming level of a light fixture by calling the *commandDimmingLevel()* method on the appropriate *LightFixture* object. This method sends the appropriate commands to an output worker thread, which in turn passes them on to the physical device.

3.6 *Persistent storage and retrieval*

The model service is backed by a relational database for persistent storage, handled by an *ObjectStorageManager* component. Each SOM object carries a unique object ID and a *ticket* obtained from a global, strictly monotonously increasing counter.

3.6.1 *Storing and retrieving single objects*

When an object's data are modified through a *set()* operation, a new ticket value v is obtained from the counter and set in the object. A new table row is then inserted into the database, containing a copy of the object as a byte array created by the Java object serialization mechanism, and the object ID and ticket value as key attributes for later retrieval. Associations to parent and child objects (which are normal Java references) are converted to object IDs before storage. To reduce delays, database storage is executed by a separate background thread. A mapping of tickets to clock 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. Due to the limited resolution of clock time-keeping (milliseconds), multiple ticket values may exist for each point in time – in this case, the highest value will be chosen. The database is then queried for the row with the given ID and the highest ticket value that is less than or equal v, and the object version is retrieved from the database und unserialized. As opposed to the latest, in-memory version of an object, older versions are immutable and cannot be changed or used to send actuator commands.

This storage scheme is based on the principles of multiversion databases (Easton 1986).

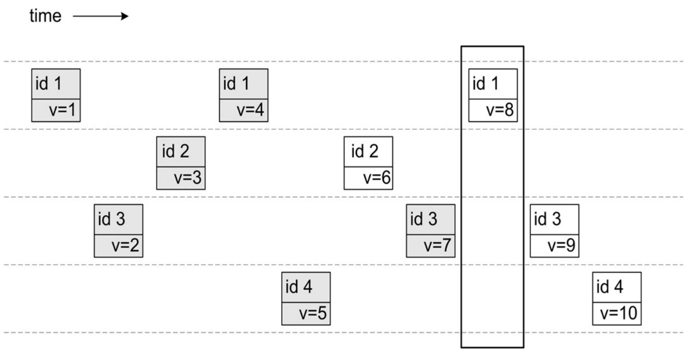

Figure 2. Four objects changing over time. At the time instant corresponding to ticket value 8, object 1 is changed. At the same point in time, the most recent versions of objects 2, 3, and 4 had the tickets 6, 7, and 5, respectively.

3.6.2 *Storing and retrieving multiple objects*

Certain operations must be grouped atomically to avoid inconsistent tree states. E.g. if a SOM object is moved from one space to another, it must be detached from the original parent object and attached to the new one, creating an intermediate step when the object is either not attached to any space, or attached to both spaces at the same time. To avoid this situation, a sequence of operations can be grouped in one ticket.

Obtaining a consistent view of a sub-tree from an object tree that is concurrently changed by other tasks (updating object data or adding or removing new objects) usually requires special measures. One way of ensuring that the object tree does not change while a client operation is traversing it is *locking*: entire parts of the tree are temporarily blocked from changes until the reading task has finished, resulting in frequent delays when many tasks are accessing regularly changed subtrees.

The storage and retrieval method outlined above offers an alternative approach that follows naturally from the single-object retrieval procedure, in the pattern of an overlapping tree (Burton et al. 1990). Assume that an application wants to traverse a subtree beginning with the root object O_1, iterating through its child objects recursively. To obtain the state of the subtree at a given time instant, it must select a ticket value v_{max} that serves as the upper bound of all ticket values in the subtree. To obtain the most recent version of the subtree (at the time of beginning the traversal), it can query the ticket counter for its current value. From then on, it recursively queries O_1 and its child objects based on the condition that the ticket value of each object must not be greater than v_{max}. If an in-memory object does not fulfil this condition (because it has changed in the meantime, resulting in a new version), the latest version with a ticket value less than or equal v_{max} is restored from the database. An example is illustrated in Figure 2.

4 EVALUATION

The aim is to test *i)* whether the chosen design and implementation work correctly so that the general requirements for a dynamic building model are fulfilled and *ii)* to test the performance of the system in a small-scale setup to estimate its scalability potential to larger setups. An overview of the system and some of the main modules used in our project is given in Figure 3.

4.1 *Experimental setup*

We are monitoring and controlling lighting in an office space with two workspaces, equipped with two dimmable uplights and two motorised Venetian blinds.

Indoor and outdoor sensor data are collected by LabView applications; actuator commands and status information pertaining to light fixtures and the motorised shading are communicated via the LUXMATE bus. A location sensing system tracks the positions of furniture and light fixtures with attached optical markers (Icoglu et al. 2004) and supplies its data over a TCP/IP connection. Additionally, the sky luminance distribution is measured through the use of a digital camera (Mahdavi and Spasojević 2004). All these data are collected and communicated as objects through a *data space* that decouples data producers and consumers.

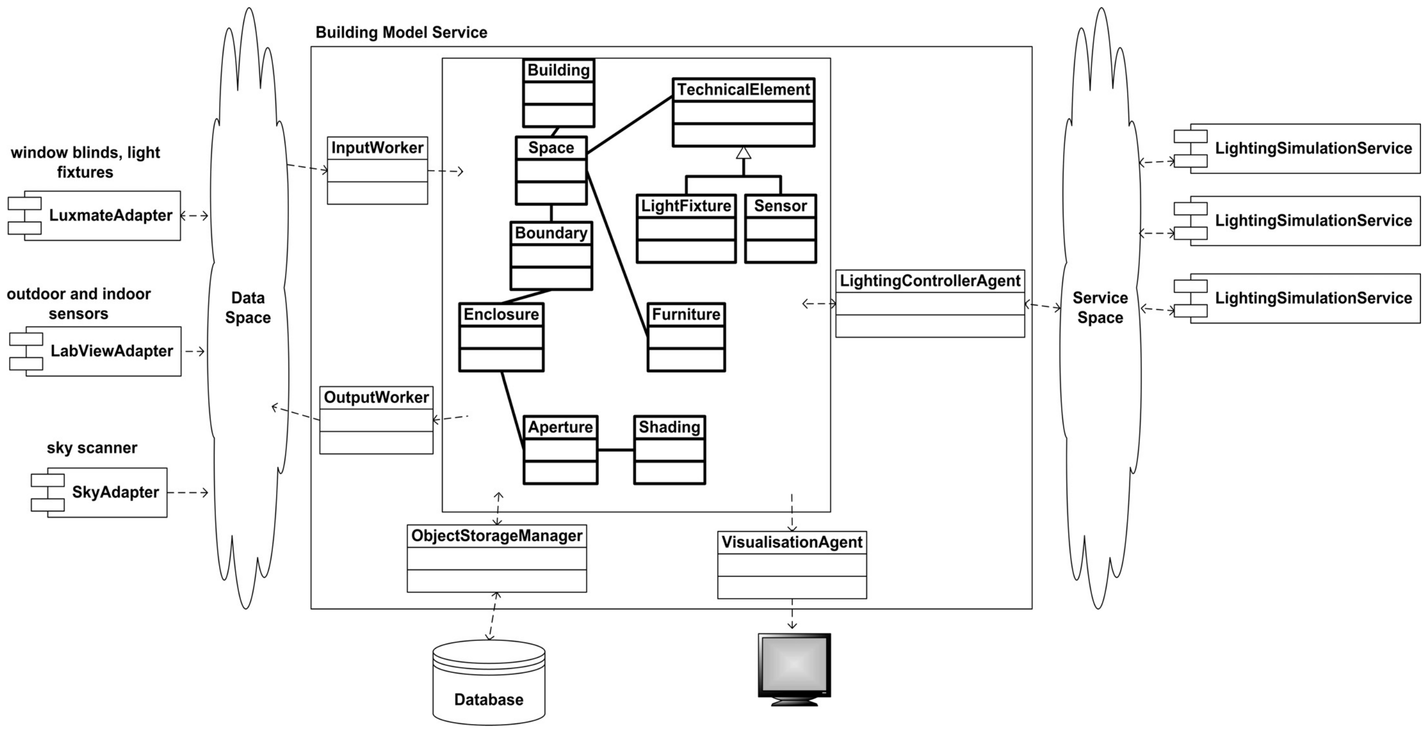

Figure 3. Overview of the building model service and its context within the experimental setup (simplified).

A commercial JavaSpaces implementation (GigaSpaces) is used for the data space and service space. The runtime locations of modules communicating over the spaces are fully transparent: any module can be run on any connected computer without configuration or code changes.

The lighting simulation package RADIANCE (Ward 1994) is used to evaluate the effects of control decisions regarding shading devices and light fixtures through a thin service "wrapper" that handles JavaSpace communications. Multiple instances of this service are started to achieve transparent processing load distribution. However, the granularity of this scheme is limited to one full simulation run: distributing the load of one simulation over multiple computers is only possible if the simulation package itself supports this.

A visualisation module based on Java3D is used to show the model's current geometric configuration as a three-dimensional rendering. It is implemented as an agent that uses the event registration facility to update itself as soon as geometry changes in the monitored objects are registered.

The model service, the relational database (PostgreSQL 8.0) and the JavaSpaces (GigaSpaces) are executed on a dedicated server equipped with an AMD Athlon MP 2000 processor and 1 GB of RAM.

The lighting controller is implemented as an agent that periodically evaluates the current lighting situation, selects a fixed number of possible control scenarios (i.e. changes of shading and electric lights) and commissions one simulation for each scenario. Simulation results are then ranked according to a utility function to select and execute the control decision. The lighting control algorithm is outlined in more detail in (Mahdavi et al. 2005).

4.2 *Results*

The full model of our experimental space is represented by about 80 objects during runtime. On average, about 30 sensor readings are arriving per second (with occasional peaks of up to 100), resulting in the same number of changes to various object attributes and database write operations.

Latency (the interval between the instant of reading a measurement from the sensor by e.g. LabView and the instant it becomes available in the model service for client applications) has been consistently below 500 milliseconds, including network communication overheads. CPU load on the model service host (defined as the percentage of processor time available to an idle priority process) remained below 15 percent on average. The model service has cumulated sensor data over the course of months reliably, resulting in close to 8 Gigabytes of database records. History queries on single objects (retrieving the state of an object at a given point in the past) take about 1 second on this database.

On our available hardware, lighting simulations take between 10 to 20 seconds each, depending on the level of input detail and simulation parameters. Batch lighting simulation requests of 16 simulations each were processed by up to 4 connected machines with RADIANCE installations. The load distribution has worked well, cutting total simulation times in direct proportion with the number of participating machines.

5 FURTHER WORK

Our building model service and applications are still a work in progress. While small-scale tests have been successful, performance and reliability tests using an actual or simulated large building are desirable to further evaluate and refine the chosen design. We are planning to build a framework for profiling and simulating varying loads in order to derive a formal performance model of the system and its key components.

The model service currently lacks a spatial indexing mechanism and offers only rudimentary spatial queries on the model. We are investigating suitable spatial indexing methods and the use of a spatial function library to add these important features. Work on the reconstruction of space boundaries from location information supplied as “tag” locations is to be integrated into the model service (Suter et al. 2005).

Retrieving snapshots of model subtrees with many frequently changing objects (according to the procedure outlined in 3.6.2) can be inefficient, as many database accesses to retrieve just recently changed objects may be necessary. We are working on a cache layer between the current model and the database that keeps recently changed object versions in memory for fast access.

The current design does not address any security aspects, relying on existing network infrastructure for access control: any system able to connect to the JavaSpace can submit an agent to the model service and change model data. In a real-world scenario, various access restrictions on the model service (such as object ownership and capabilities) would be necessary.

6 CONCLUSION

We have outlined the requirements for a dynamic building model service, and have described a prototype design and implementation. An experimental application running simulation-based lighting control for an office space has been implemented. Our preliminary evaluation has shown that the chosen design is feasible and has the potential for being tested in larger-scale configurations.

While originally driven by the requirements for simulation-based control, we expect dynamic building model services to be a valuable foundation for a wide range of different applications in building operation and management.

ACKNOWLEDGEMENT

The research presented in this paper is supported by a grant from FWF (Austrian Science Foundation), project number P15998-N07. The research team includes, in addition to the authors, G. Suter, O. İçoğlu, B. Spasojević, and J. Lechleitner. The authors gratefully acknowledge comments and suggestions from G. Suter.

REFERENCES

Brunner, K.A. & Mahdavi, A. 2005. A software architecture for self-updating life-cycle building models. In press: *Proceedings of CAADfutures 2005*, Vienna, Austria.

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

Bushby, S.T. 1997. BACnet - a standard communication infrastructure for intelligent buildings. *Automation in Construction* 6(5-6):529-540.

Clarke, J.A., Cockroft, J., Conner, S., Hand, J.W., Kelly, N.J., Moore, R., O'Brien, T. & Strachan, P. 2001. Control in building energy management systems: the role of simulation. In *Proceedings of the Seventh International IBPSA Conference, Rio de Janeiro*: 99-106.

Eastman, C.M. 1999. *Building Product Models: Computer Environments Supporting Design and Construction*. Boca Raton: CRC Press.

Easton, M.C. 1986. Key-sequence data sets on indelible storage. *IBM J. Res. Dev.* 30(3): 230-241

Freeman, E., Hupfer, S. & Arnold, K. 1999. *JavaSpaces Principles, Patterns, and Practice*. Boston, Mass.: Addison-Wesley.

Guillemin A. & Morel N. 2001. An innovative lighting controller integrated in a self-adaptive building control system. *Energy and Buildings* 33(5):477-487.

Icoglu, O., Brunner, K.A., Mahdavi, A. & Suter, G. 2004. A distributed location sensing platform for dynamic building models. In Markopoulos, N. et al. (eds.) *Ambient Intelligence: Proceedings of the Second European Symposium, Eindhoven (Lecture Notes in Computer Science 3295*): 124-135. Berlin: Springer-Verlag.

Lam, K. P., Wong, N. H., Shen, L. J., Mahdavi, A., Leong, E., Solihin, W., Au, K. S. & Kang, Z. J. 2003. Mapping of industry building product model for thermal simulation and analysis. In *Proceedings of the Eighth International IBPSA Conference, Eindhoven* Vol. 2: 697-704.

Luxmate Controls GmbH. 2005. Internet resource URL http://www.luxmate.com (accessed 25 April 2005).

Mahdavi, A. 1997. Toward a simulation-assisted dynamic building control strategy. In *Proceedings of the Fifth International IBPSA Conference*. Vol. I: 291-294.

Mahdavi, A. 2001. Simulation-based control of building systems operation. *Building and Environment*. 36(6): 789-796.

Mahdavi, A. 2004. Self-organizing models for sentient buildings. In: Malkawi, A. M. & Augenbroe, G. (eds.) *Advanced Building Simulation*: 159 – 188. London: Spon Press.

Mahdavi, A. & Spasojević, B. 2004. Sky luminance mapping for daylight-responsive illumination systems control in buildings. In *Proceedings of the 35th Congress on HVAC&R, Belgrade, Serbia and Montenegro*: 93 – 102.

Mahdavi, A., Ilal, M.E., Mathew, P., Ries, R., Suter, G. & Brahme, R. 1999. The architecture of S2. In: *Proceedings of Building Simulation 1999, Sixth International IBPSA Conference Kyoto, Japan*, Volume 3: 1219-1226

Mahdavi, A., Spasojević, B. & Brunner, K.A. 2005. Elements of a simulation-assisted daylight-responsive illumination systems control in buildings. In press: *Proceedings of the Ninth International IBPSA Conference,* Montréal, Canada.

Pargfrieder, J. and Jörgl, H.P. 2002. An integrated control system for optimizing the energy consumption and user comfort in buildings. In *Proceedings of the 2002 IEEE International Symposium on Computer Aided Control System Design*: 127-132.

Sharples, S., Callaghan, V. & Clarke, G. 1999. A multi-agent architecture for intelligent building sensing and control. *International Sensor Review Journal*, 19(2): 135-140.

Suter, G., Brunner, K. & Mahdavi, A. 2005. Spatial reasoning for building model reconstruction based on sensed object location information. In press: *Proceedings of CAADfutures 2005*, Vienna, Austria.

Ward, G.J. 1994. The RADIANCE lighting simulation and rendering system. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques:* 459-472.