Report

An Interlink of Building Control System Prototypes and the Lighting Simulation Lumina

Andreas Metzger

Working Group VLSI Design and Architecture

Department of Computer Science
University of Kaiserslautern

August 1999

Department of Computer Science

University of Kaiserslautern · P.O. Box 3049 · D-67653 Kaiserslautern

Contents

Section 1 Introduction

When designing a building to be energy and resource efficient, yet providing user comfort and safety, the building's overall performance has to be optimized in the early design stages [Aug92, AuW91]. The **physical structure** of the building (floor plan, used materials, etc.) as well as the **building control system** affect this performance.

To evaluate the consequences of physical design decisions, **simulations** of the planned building have to be utilized. The results of these simulations can then be used to iteratively refine the design in further cycles.

A promising integrated, component based approach for designing and simulating buildings is currently under development at the Built Environment Research Laboratory of the School of Architecture at Carnegie Mellon University, Pittsburgh, PA. The project, called **SEMPER**, and its lighting simulation module **Lumina** are introduced in sect. 1.1.

Building control systems allow further performance optimizations. Because it cannot be guaranteed that the building will be used as intended during its whole life span, such a control system must be flexible and easily adaptable to changing requirements.

A modeling method for building control systems is introduced in sect. 1.2. The method was developed at the Computer Science Department of the University of Kaiserslautern, Germany. Control systems are specified using hierarchical concepts to deal with such a system's complexity. The underlying modeling language is employed to generate executable programs that are used as prototypes of the final system. These prototypes can be used to test the control system and to present the functionality of the future system to the users.

Building control systems and their **environment** are closely interlinked. Therefore, prototypes of the control system need to be connected to building simulators (or to the physical building in later design stages) to produce feasible tests.

Fig. 1: Interlink

The control system prototypes and their environment can differ in several aspects; e.g. the interfaces and communication protocols can be dissimilar (refer to section 2 for further aspects). Using an interlinking component—or an **interlink** for short—we are able to establish a connection (refer to fig. 1).

The results of a conjoined run of both, the building simulator and the control system prototype, deliver important feedback not only for the control system design but also for the design of the physical building.

1.1 SEMPER and Lumina

We want to test control system prototypes using the simulation modules of SEMPER. In this paper we implement an interlink between a lighting control system prototype and SEMPER's lighting simulation module Lumina.

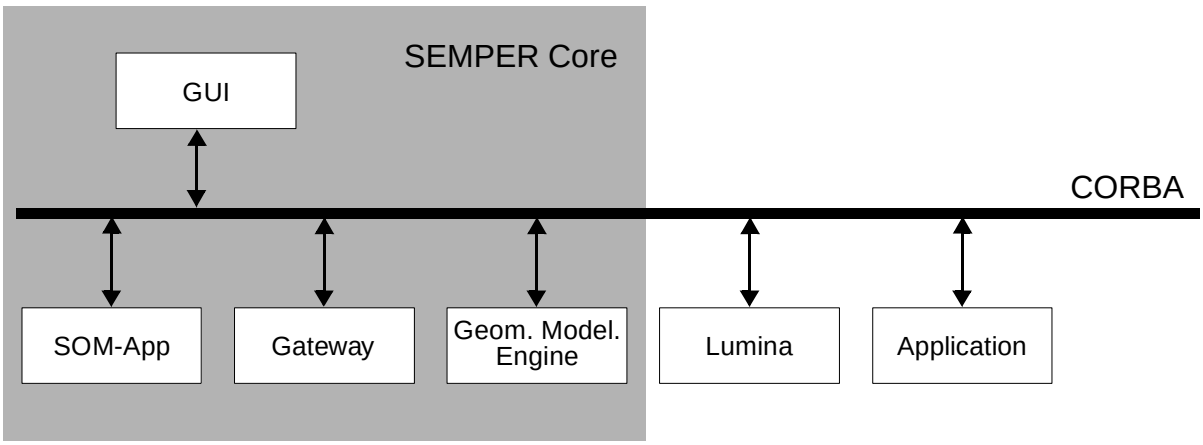

Fig. 2: SEMPER's Component Architecture [MMB98]

Lumina is integrated into SEMPER as shown in fig. 2.

The SEMPER core's main data basis is a space-based object model of the building, the **shared object model** (**SOM**). It contains objects like sections, spaces, exterior elements and enclosures.

SOM-App ('SOM-Application') contains the SOM and provides special methods to modify objects in the SOM. These methods rely on the **Geometric Modeling Engine** for geometric mod-

ifications. In addition, SOM-App contains the functionality to start simulations and to return results.

The **GUI** allows the user to interact with SOM-App through different user interfaces. Floor plans can be edited, semantic information for spaces can be entered, and spaces can be assigned to zones. Every change in the physical structure maps directly to a change in the SOM.

The **Gateway** component is used to initially connect the GUI with SOM-App.

Different **Applications** can communicate with the SEMPER core using CORBA (Common Object Request Broker Architecture [OMG98]). Typically, these applications are simulators for specific domains; e.g. lighting or heating. The **domain object model** (**DOM**) that is used as input to such a simulator is derived from the SOM. [MMB98, MMB99, Mah96, MML96]

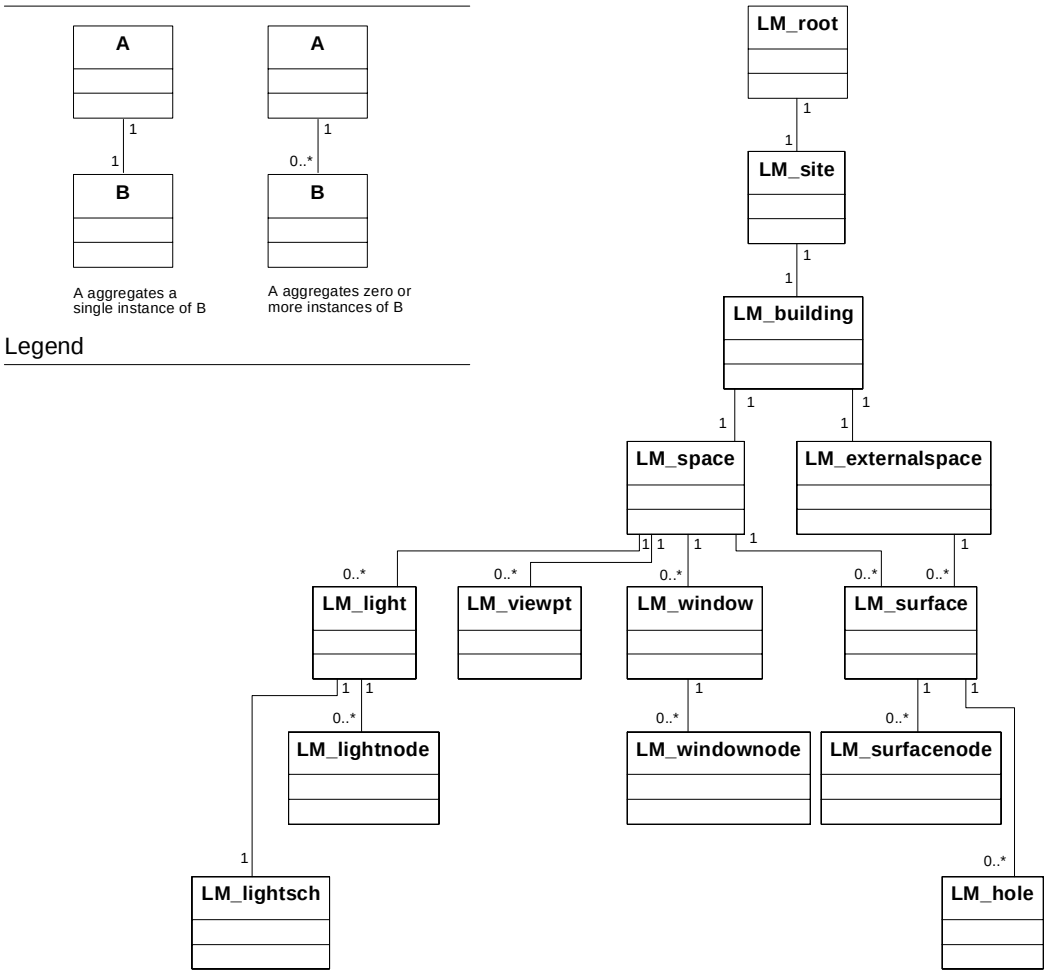

Fig. 3: Lighting DOM [PaM99]

Lumina

For the lighting simulation, a component called Lumina [PaM99] exists. Fig. 3 shows the object model used by Lumina.

In the lighting DOM the building is composed of spaces that are made up of surfaces. Surfaces are described by surface nodes that describe the corners of the surface, and holes that describe openings in the surface. Internal and external spaces are distinguished. Internal spaces may contain windows (described by window nodes), lights, and viewpoints of ‘virtual’ persons. The position of the lights is contained within light nodes, whereas light-levels are contained within light schedule objects.

Daylight is computed using a discretized sky model. The sky model is derived from information about the building site (geographical location, orientation, etc.). Radiosity methods are used to calculate the diffuse components of the illuminance [HeB94].

1.2 Specifying Building Control Systems

We describe complex systems using a formal specification language rather than a programming language. Program code exposes too many details. Therefore, much time is devoted to optimizing code and finding programming errors instead of concentrating on a system’s functionality. By using a formal specification language, these errors can be avoided by describing the structure and behavior of the system at a higher level. Further, if the chosen specification language allows hierarchical or object oriented concepts, complexity can be dealt with more easily. In addition, formal specification languages allow verification based on mathematical methods.

By employing an **executable** specification language we are able to generate **prototypes** automatically. Prototypes can be used, e.g., to present the functionality of the future system to the customer. Therefore, **prototyping** supports the early phases in system development by helping to identify customer requirements [Met98]. Also, prototypes can be used by developers for **verification** (through testing) or for **validation** [Jal97].

SDL[1] [Z.100] is chosen as a specification language. The existing tool SDT[2] from Telelogic [Tel99] is used.

Emphasis is laid on an efficient modeling method that can handle large building control systems. A modern building contains a huge number of controllable devices that are interconnected. To give an example, a team of researchers of the SFB 501 at the University of Kaiserslautern specified a control system of an office floor with 22 offices. This control system makes use of 578 sensors and actuators controlled by 360 controllers [SFB99].

1. Specification and Design Language [Z.100, OFM94]
2. SDL Tool [Tel99]

Modeling Method and Templates

Our modeling method is supported by a few **modeling templates**. Because of the small numbers of templates the method can be learned and employed very fast. The hierarchical structure aids in finding errors and allows gaining an overview of the whole system very easily [Zim98, Que98].

Our modeling method produces an **organizational hierarchy**, where, like in large organizations, the system's behavior is distributed over all levels. The behavior that is specified for a level matches the tasks that can be performed at that level. This kind of partitioning allows minimizing the flow of information.

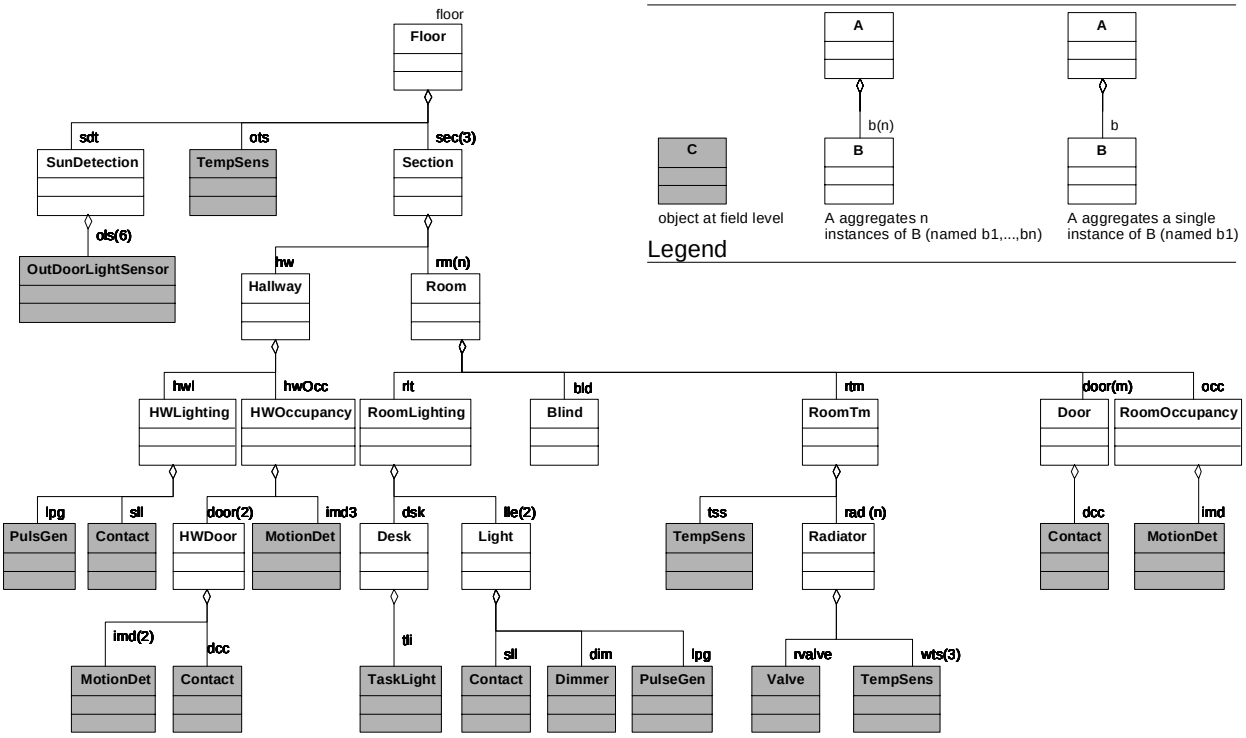

Fig. 4: Organizational Hierarchy of an Office Floor [SFB99]

Figure 4, which shows the object model of the above example, is employed to illustrate this idea. Sensors and actuators are located at field level and are modelled as **leaves** in this hierarchy (shown as shaded boxes in figure 4). More complex controls are found at higher levels; for example, temperature or lighting controls. The top-level object contains facility management functionality.

Communication is allowed only between an object and its direct sons. Therefore, if objects on different branches need to communicate, the signals from the sending object have to be propagated to the object where the branches meet. From there the signals are passed down through the tree to the receiving object. This simplifies the reuse of objects and the overall modeling process because changes affect objects only on a local scale.

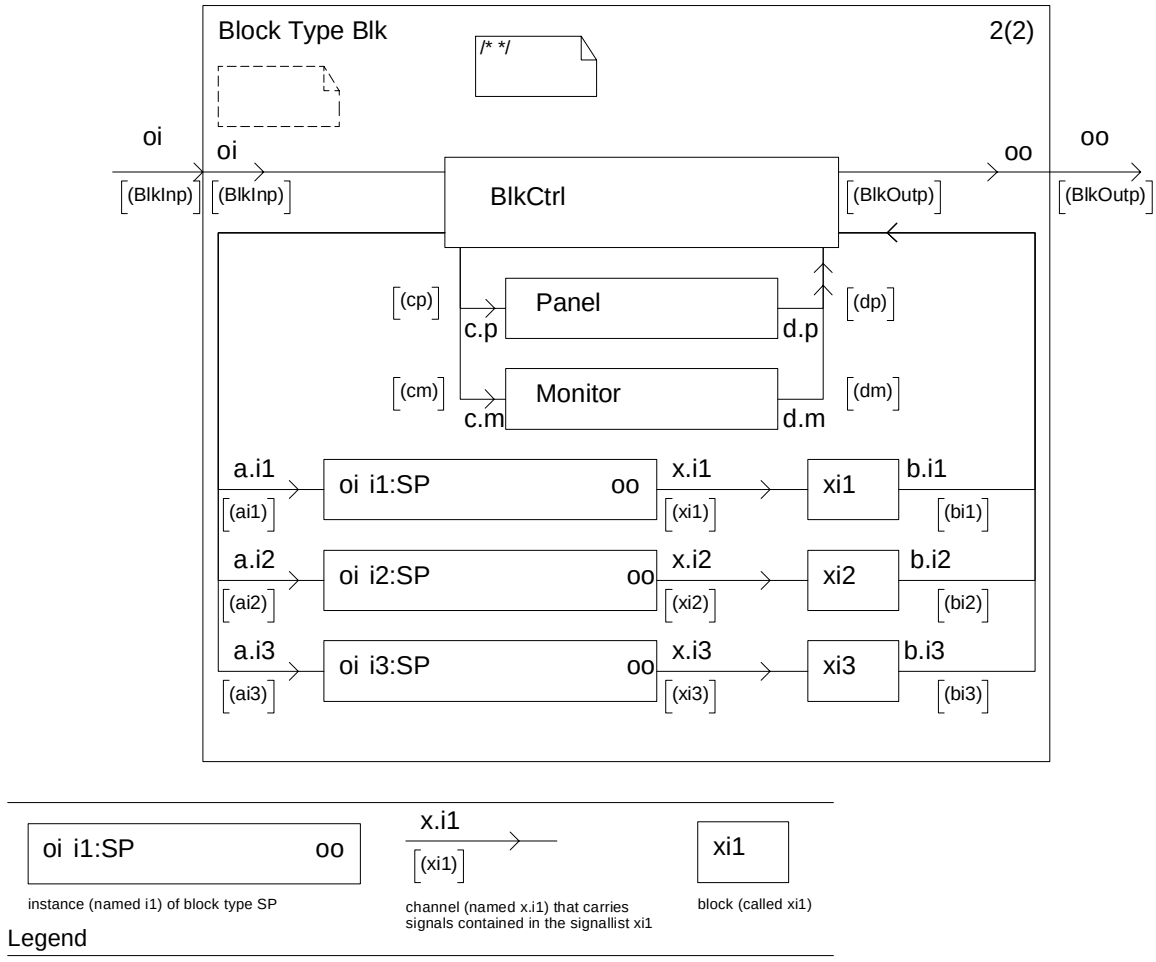

Fig. 5: Block Type Modeling Template [Zim98]

An object in such a hierarchy can be specified using the block type modeling template (see fig. 5). This template consists of a **controller block** (`BlkCtrl`) and further **sub-objects** (instances of the type `SP`), which communicate by exchanging signals[1]. All signals have to pass through `BlkCtrl` because direct communication between sub-objects is not allowed (see above).

Objects and sub-objects are instances of SDL block types. `SP` is replaced by the name of the block type of the respective sub-object. Because SDT doesn't support parametrized types yet, the blocks `xi1`, `xi2`, and `xi3` are used to uniquely identify instances (refer to [Que98] for details).

To apply this modeling pattern, each occurrence of `Blk` is replaced with the name of the component type; e.g., `Light`. Instances of `SP` are replaced by instances of types aggregated by this com-

1. SDL signals may contain additional data.

ponent; e.g. `Luminaire` and `IlluminanceSensor`. Also, the instance names (`i1`, ...) should be changed to reflect the object's function; e.g., `lgt1`, `ils1`. See fig. 16 in section 3 for an example.

The block `Panel` is used as a connection point to physical panels that are typically simulated by software components during the prototyping process. Panels are employed for user interaction; for instance, a panel might be utilized to enter the set-point for lighting control, or it might be employed to signal certain system conditions to the user.

Within `Monitor`, a simple data-logger can be implemented. In its simplest form, `Monitor` writes data to a file.

`BlkCtrl` contains the object's behavior described by **extended finite state machines**. Finite state machines consist of states and transitions connecting the states. A transition is triggered by a signal or a timer event. It can produce an output when triggered. The history of the finite state machine is expressed by its state. An extended finite state machine, additionally uses variables to maintain its history [OFM94].

Naming the Objects

To get unique names (**instance-names**) for objects in our system, we name every block instance and concatenate these names when going up to the root of the object hierarchy. Fig. 4 is used to illustrate this. The second temperature sensor of the first radiator in room number 8 in section 2 of our office floor has the instance-name `wts1rad1rtm1rm1sec2floor1`.

After the control system has been specified, a prototype can be generated. This prototype is connected to its environment using sockets[1] and a simple protocol. Sensor data is returned as a character string containing the instance-name of the sensor and the sensor data; e.g., `"wts1rad1rtm1rm1sec2floor1 20.5"`. Actuators can be controlled by emitting commands that contain the instance-name of the actuator, followed by `"setValue"` and the desired set-value; to illustrate, the radiator valve of the above radiator can be opened with `"rvalve1rad1rtm1rm1sec2floor1 setValue 1"` (refer to [MQS96] for details).

1. Internet stream sockets, which are connection oriented, bidirectional communication channels based on the TCP/IP protocol stack, are used [Rag93].

Section 2 The Interlink

The control system prototype and the building simulator can differ in several aspects. These aspects and how an interlink can resolve the problems, originating from the differences, is presented in the following paragraphs.

- If the two systems use incompatible **communication protocols**, and if changing the communication method presents too much effort, an interlink that converts messages of one protocol to messages of the other protocol is needed.
- A system's object structure can change during the development process. Therefore, its **object names** (e.g., instance-names in the control system) may vary. A respective modification of the structure in the other system to match these changes might not be possible or reasonable. Introducing an interlinking component allows a rather effortless mapping of objects from one system to objects in the other system.
- The **specification** or **programming language** can be different. Merging the systems into one and eliminating the need of 'external' communication protocols is not conceivable if reprogramming one of the systems is too costly.
- The two systems can handle **time** differently; e.g., where the control system may perform its actions in real-time, the building simulator might need several minutes to calculate results. An interlink can be used to synchronize the two systems.
- Figure 6 (on page 13) shows how the control system prototype and the environment can be partitioned.

  The control system prototype can consist of multiple **partitions**—possibly residing on different machines. Partitioning is needed if the final implementation of the control system should consist of smaller systems that are running on individual controllers. Distributed prototyping allows evaluating the distributed system, and thus aids in finding an optimal partition. An

interlinking component connects the partitions with the environment. Further, it can be employed to synchronize the prototype partitions and enable communication between them

If the environment is simulated by **more than one simulation module**, or objects in the building simulation are replaced by real world objects for **hardware-in-the-loop** tests, an interlinking component can be used to connect these modules to the control system prototype. An interlink dispatches control system messages to the correct simulation modules or the building and vice versa.

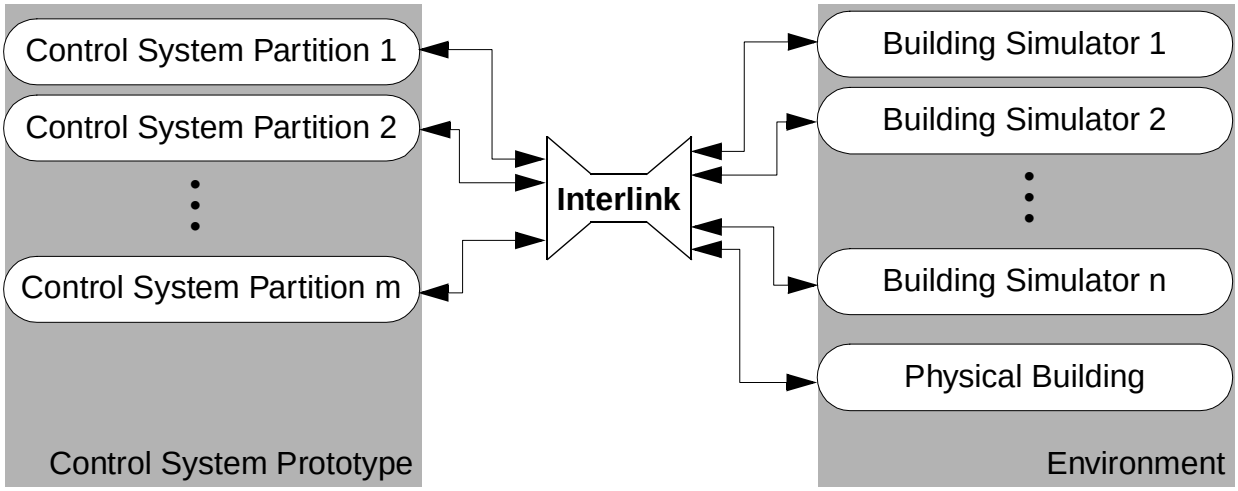

Fig. 6: An Interlink of Control System Partitions, and Simulation Modules and the Physical Building

For our case, the differences between the control system prototype and the lighting simulator Lumina are shown in the following table.

Aspect	Control System Prototype	Lumina
protocol	TCP/IP sockets	CORBA
object names	unique instance-names	CORBA object references, indices
specification/programming language	SDL	C++
time	real-time	discrete time-steps (computing time approximately one minute per time-step)

Table 1: Comparison of the Control System Prototype and Lumina

2.1 Lumina and CORBA

The communication protocol that is employed by Lumina is CORBA. The following paragraphs present a short overview of CORBA, followed by an explanation of how CORBA is integrated into the lighting simulator.

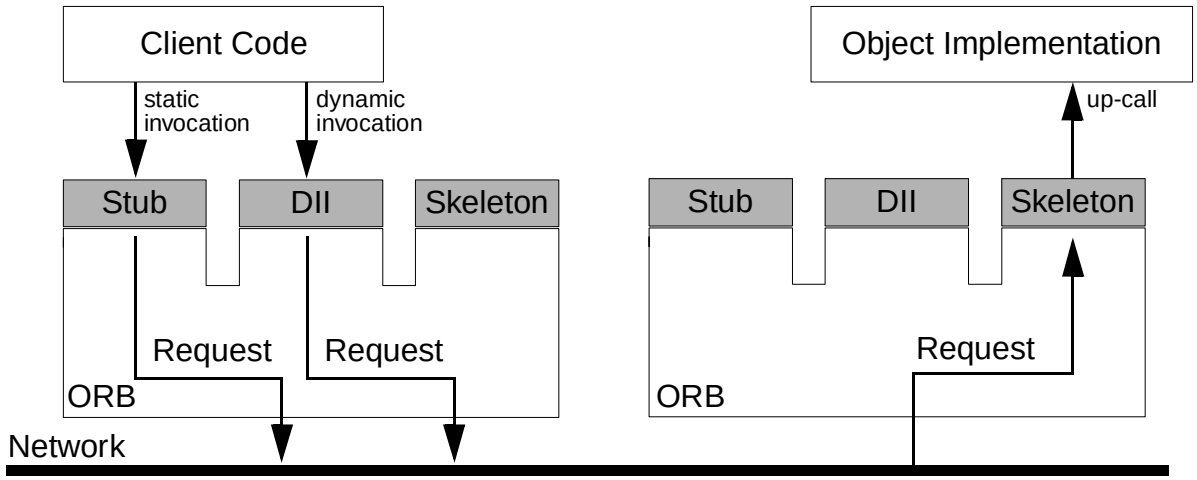

Fig. 7: A Request Being Sent Through the ORB [Sie98, OMG98]

The Common Object Request Broker Architecture (CORBA [OMG98]) defines mechanisms that allow the transparent object interaction in a distributed computing environment. Objects communicate with each other through an **ORB** (**Object Request Broker**) as shown in fig. 7. An ORB connects objects requesting services to objects providing the services.

The interface of an object is defined using **IDL** (**Interface Definition Language**). After the interface has been defined, the **object implementation** and the **client code**[1] can be developed independent of each other.

Several IDL mappings to different programming languages exist, including Java, C++ and Smalltalk. Language specific **stubs** and **skeletons** are generated using IDL compilers. Stubs connect the client code with the ORB. Skeletons connect the ORB with the object implementation.

Objects are accessed using CORBA object references. Object references can be returned by other CORBA objects or they can be retrieved using the **Naming Service**. The Naming Service acts as a dictionary where object references can be looked up by their unique name.

After an object reference has been retrieved, an object's method can be called using two different schemes. The **Static Invocation Interface** (**SII**) is applied if the method names and the formats

1. "The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object" [OMG98]

of the parameter lists are known before run-time. The stubs on the client side implement this kind of invocation interface.

If the method names are unknown before run-time, the **Dynamic Invocation Interface** (**DII**) can be applied. Method names and parameter lists are constructed during run-time and are passed to the ORB using special DII methods.

2.1.1 CORBA Objects for Sensors and Actuators

The actual connection points between the control system and its environment are **sensors** and **actuators**. Lumina's domain object model (DOM, refer to fig. 3) is extended by CORBA objects that represent sensors and actuators as described in the following paragraphs.

Sensors are used to determine the physical state of the building (e.g., temperature or illuminance), whereas actuators are used to influence the physical state by driving the electrical and mechanical installations (e.g., radiator valves, luminaires or air conditioning fans).

Two types of actuators can be distinguished in Lumina: **geometry-based actuators** and **non-geometry-based actuators**.

Geometry-Based Actuators

Geometry-based actuators influence the physical state of a building **by changing the geometry** of associated objects. Louvers, blinds and automatic doors are typical examples. If the control system sends a signal to such an actuator, the geometry of the objects will change.

An actuator of this type is represented by a CORBA object that contains a method called `setValue()` together with references to the geometric DOM objects that are to be modified. The method requires a discrete value (e.g., angle of the louver) and calculates the new coordinates for the associated geometry objects.

Non-Geometry-Based Actuators

Non-geometry-based actuators influence the physical state **without changing the geometry**. Luminaires are a good example, values from 0.0 to 1.0 can be mapped to light-levels from 0% (luminaire is turned off) to 100% (full brightness).

If actuators influence the physical state on a scale not visible to the simulation or if this fact is not important for the control system, these actuators should be regarded as non-geometry-based actuators. Typical examples are radiator valves and vents.

Non-geometry-based actuators are represented by CORBA objects that contain a method called `setValue()` that requires a discrete value from which the state of the actuator is calculated.

The distinction between geometry- and non-geometry-based actuators can help speeding up the simulation. Lumina uses radiosity methods [HeB94] to calculate the diffuse component of the illuminance. Form factors have to be recalculated if the geometry of the scene has changed. If

only non-geometry-based actuators have been accessed during a simulation step, the calculation of the form factors can be skipped.

Sensors

Simulation results are stored in CORBA objects that represent sensors that provide this data. All sensor objects contain a method called `getValue()` to retrieve this information from the client side.

Generic Approach

Using inheritance, the above concepts can be realized in a generic manner that allows an easy extension and modification. Figure 8 presents an overview. All sensor objects inherit from `CX_sensor`, whereas all actuators inherit from `CX_actuator`.

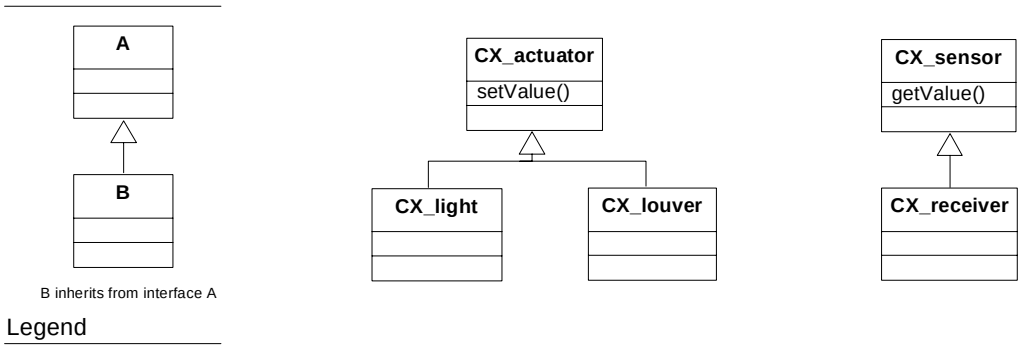

Fig. 8: Object Model of CORBA objects

Each specialization of `CX_sensor` has to implement `getValue()`. Every specialization of `CS_actuator` has to implement `setValue()`.

Figure 9 (on page 17) shows the realization of the above concepts for Lumina. The DOM is extended by CORBA objects that represent sensors and actuators. Further, a root object `CX_root` is introduced that aggregates all sensors and actuators, and provides functions to retrieve object references (refer to section 2.1.2 for details).

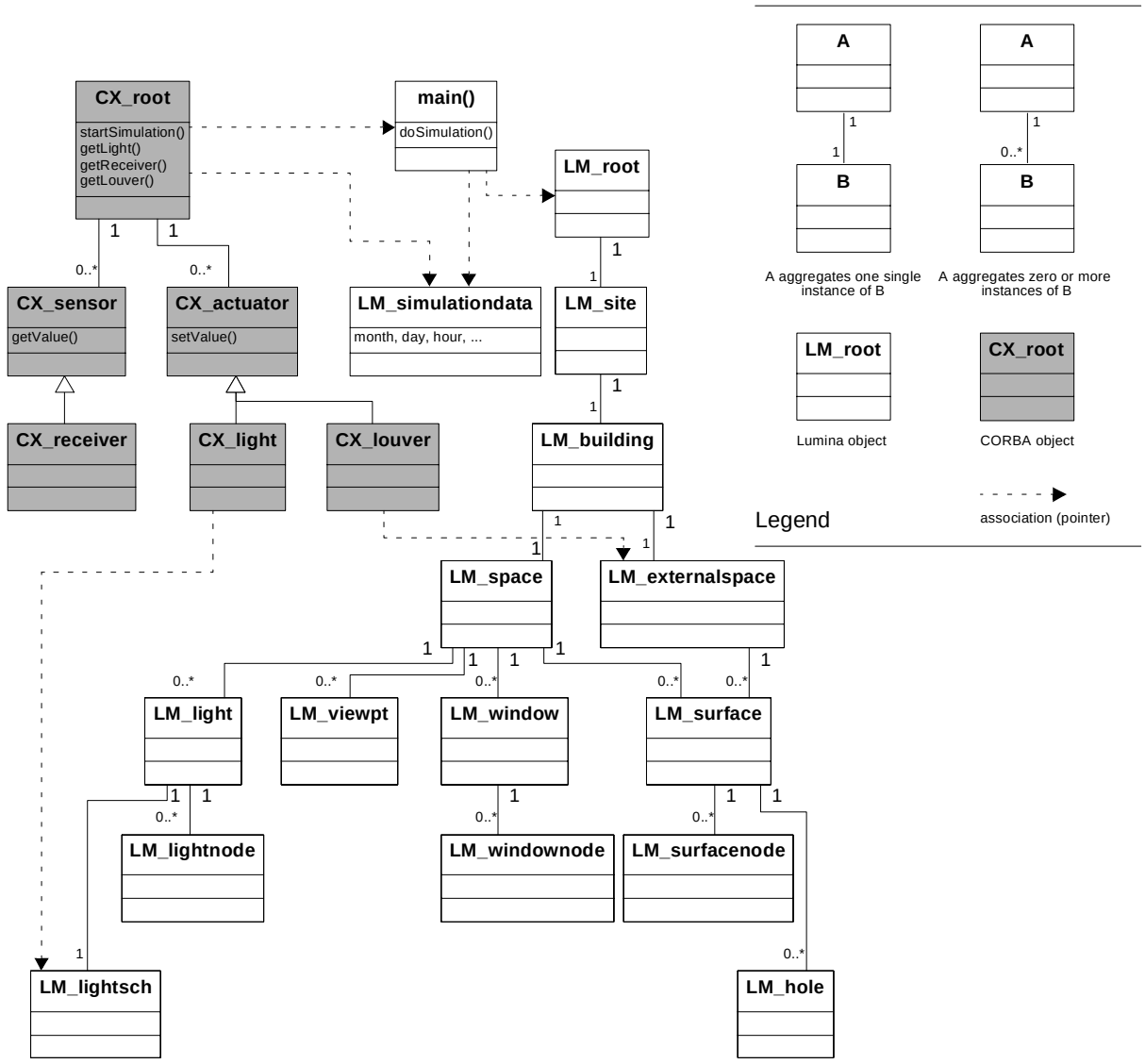

Fig. 9: Lighting DOM with CORBA Extensions

Methods can be added to implement additional functionality. In our case, methods have been added to assign pointers to non-CORBA objects (existing code). This realizes a simple form of delegation. A request to a CORBA object is delegated to a non-CORBA object using pointers (shown as dashed arrows in fig. 9).

To illustrate, if the `setValue()` method of `CX_light` is called, the respective value in the `CX_lightsch`[1] object is modified by following the pointer to this object.

1. The `CX_lightsch` object maintains the light-level of the luminaire.

2.1.2 Retrieving Object References

CORBA object references are returned by methods of the root object `CX_root` in the CORBA object hierarchy. We call these methods **visitor functions**. They require an index of the object and return a CORBA object reference.

Visitor functions retrieve object references by employing Lumina's object structure. This frees the client applications from knowing the DOM's structure. Therefore, a change in the DOM doesn't have any consequences on other components as long as the indexing (or naming) scheme isn't altered.

In Lumina's CORBA implementation visitor functions are of the form `get<TypeName>`, where `<TypeName>` is replaced by the requested object's type; e.g. `Louver`.

2.2 Synchronizing the Control System Prototype and Lumina

To realize a **synchronized** run of the control system prototype and Lumina, we have to halt the control system prototype until the simulator has finished its calculations.

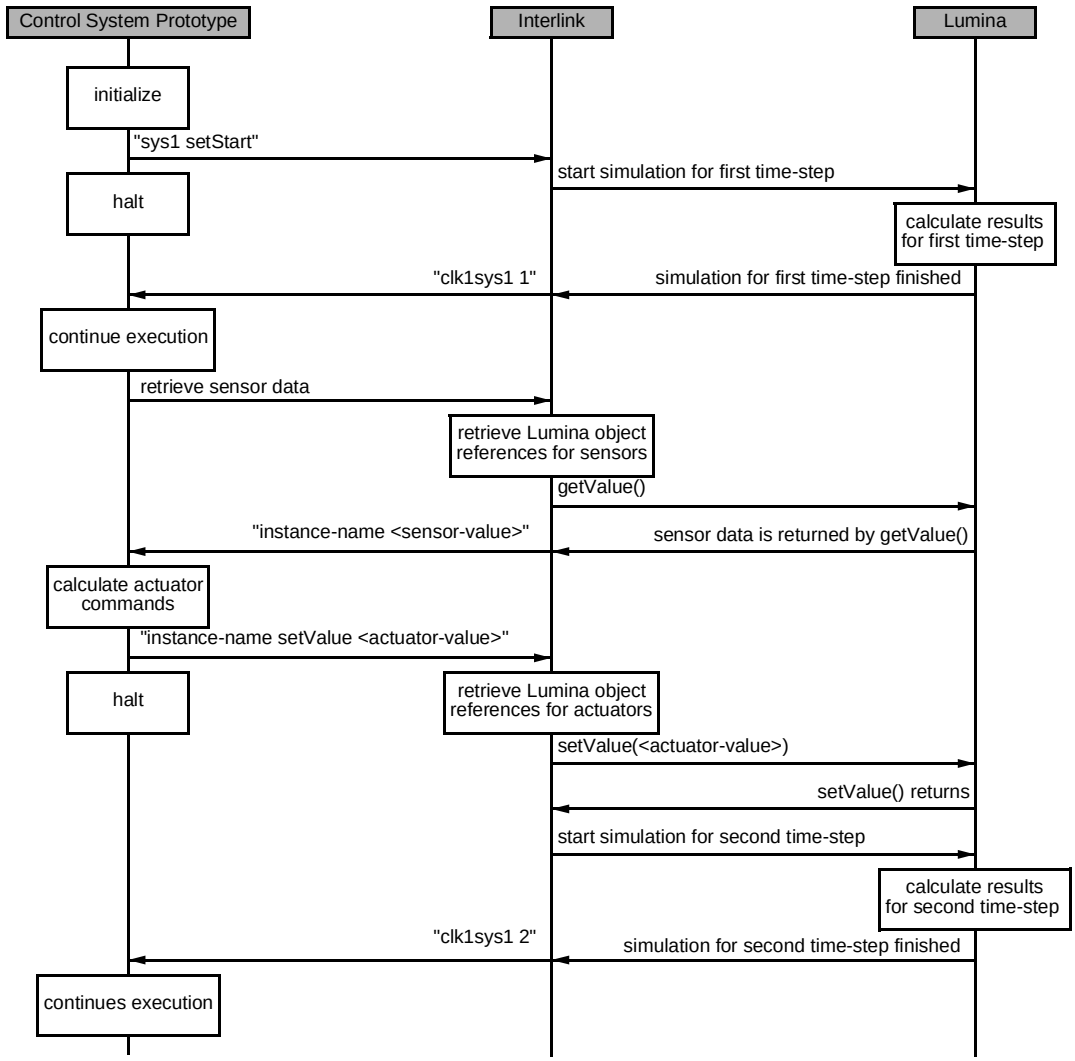

Fig. 10: Diagram of 'Ideal' Synchronization Mechanism

Figure 10 shows how the control system prototype and Lumina communicate 'ideally' using the interlink.

The control system prototype needs some time to initialize. We can determine the end of the initialization phase easily and can send a message (`"sys1 setStart"`) to the interlink to start the simulation. Until the results for the first time-step arrive, default values are used for sensor data. Alternatively, the control system can be halted until results are available.

After the initialization, the control system waits for the lighting simulation to finish calculating results for one time-step. The interlink signals this to the control system using the message `"clk1sys1 <n>"`, where `<n>` stands for the calculated time-step. It then retrieves sensor data (`"<instance-name> getValue"`) and calculates reactions to new sensor information. Usu-

ally, this leads to emitting actuator commands. After actuator commands (“`<instance-name> setValue <actuator-value>`”) have been conveyed to interlink and thus to the lighting simulator, results for the next time-step can be calculated.

To implement the behavior shown in figure 10, not only the control system prototype must be halted but also it must be determined if the prototype has finished its calculations. Because there is no simple way of determining if all scheduled state-transitions in our control system have been performed[1] we do not implement this ‘ideal’ synchronization protocol at this point.

Instead, we assume that the calculation of one time-step takes longer than the reaction of the control system to new sensor data (in our case the control system reacts in less than one second, wheras the calculation of one time-step takes about one minute). As soon as data for a time-step has been computed, the simulator immediately starts its calculations for the succeeding time-step. While this simulation is running, the control systems reacts to the new sensor data by emitting actuator commands. These commands are stored and are used as input to the simulator after the currently running calculation has finished. Consequently, we use the results for every other time-step as sensor data. Otherwise, sensor and actuator states would not be consistent.

2.3 Modifications on the Control System Side

Our first realization of the control system prototype assumed a real-time environment. Unfortunately, Lumina cannot provide results in real-time[2]. Therefore, a few modifications for the prottype generation have to be introduced.

1. To retrieve this information we have to employ SDT’s data structures and look inside its scheduler.
2. In this context real-time means that simulation results are produced at such a rate that no difference to a physical building is recognized.

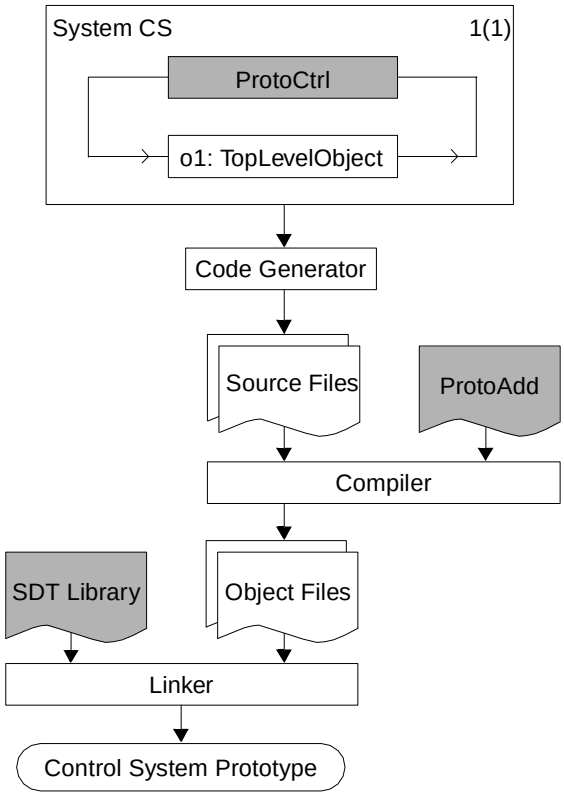

Fig. 11: Generating a Control System Prototype from an SDL Specification (SDT specific)

The Prototype Generation

The control system prototype is generated according to fig. 11. We have to introduce changes in the components **ProtoCtrl**, **ProtoAdd** and **SDT Library** to realize the above synchronization mechanism.

The control system designer specifies the control system up to the **TopLevelObject** using the templates presented in section 1.2. This object is connected to the SDL block ProtoCtrl to form an SDL system that can be used to generate prototypes from. ProtoCtrl (ProtoController) contains SDL specifications that are needed for the prototype execution. The internal details of ProtoCtrl are usually not visible to the control system designer.

After the system has been specified, C-Code can be generated using SDT's code generators.

Functions that are difficult to express in SDL can be written in C and added to the generated source files. ProtoAdd—an addendum to ProtoCtrl—contains these functions. In our case, source code that implements communication functionality, type-conversion, and file I/O is located here.

After the generated source files and ProtoAdd have been compiled using an ordinary C compiler, an executable file is produced by linking the object files with the SDT library. The SDT library

[SDT98a] contains the scheduler that handles the execution of SDL's extended finite state machines. It also contains the function `SDL_Clock()` that returns the SDL **system time** `NOW`. The value of `NOW` is used by SDL timers to trigger timer events.

2.3.1 Modifications in the SDT Library

Using the shared variable `syncSimTime`, we increment the value of `NOW` every second (2nd) time the interlink signals the control system that a time-step has been computed. The shared variable is incremented in ProtoAdd (see paragraph 2.3.2).

After all timer events for the current value of `NOW` have been executed and all state transitions that were triggered by these events have been performed, the system halts until the variable `syncSimTime` is incremented once more. Fig. 12 shows the code fragment that implements the modified `SDL_Clock()` function.

```
SDL_Time SDL_Clock  XPP(( void ))
{
     SDL_Time tmp;

     tmp.ns = (xint32)0;
     tmp.s[1]  = (xint32)syncSimTime;

     return tmp;
}
```

Fig. 12: SDL_Clock() function

2.3.2 Modifications in ProtoAdd

Our socket protocol that basically uses "`getValue <instance-name>`" to retrieve sensor data, and "`setValue <instance-name> <value>`" to emit actuator commands. We extend this protocol by adding the two new messages "`sys1 setStart`" and "`clksys1 <n>`".

After the control system prototype has been initialized (this is determined in ProtoCtrl, see paragraph 2.3.3), the socket message "`sys1 setStart`" is sent to the interlink. The interlink receives this message and starts the simulation for the first time-step.

The message "`clksys1 <n>`", where `<n>` represents the number of the time-step that has been computed, can be received by the control system prototype. ProtoAdd interprets this message and increments the shared variable `syncSimTime` accordingly.

1. SDL 'seconds' are chosen as smallest time-step.

2.3.3 Modifications in ProtoCtrl

After having introduced the modifications in the SDL library, SDL timers can only be triggered after the lighting simulation has finished a time-step and `NOW` is incremented.

ProtoCtrl periodically calls a function to check if new socket messages have arrived. This function was called with a delay of one time-step in the original version of ProtoCtrl. Because the socket message `"clk1sys1 <n>"` is used to increment the system time (`NOW`), we have to remove this delay or we won't receive any socket messages at all. The delay has been introduced for efficiency reasons and removing it has no influence on the system's functionality.

The instance-names of all objects in the control system are stored in a table in ProtoCtrl. During the initialization of the control system, each block instance's name is propagated through the tree hierarchy to ProtoCtrl. The end of this initialization process can be identified by the reception of the `ready` signal. This signal is emitted by a block if all sub-blocks have been initialized. Therefore, if ProtoCtrl receives this signal, the `"sys1 setStart"` message is sent to the interlink to mark the end of the initialization phase.

2.4 Implementation of the Interlinking Component

After the above modifications have been introduced we can finally realize the interlinking component. The following sections present the basic considerations that were taken into account for the implementation.

2.4.1 Mapping Issues

In Lumina, sensors and actuators are realized using CORBA objects. To access these objects, an unambiguous mapping from control system objects to DOM objects has to be used. An object in the control system can be referenced by its instance-name. Objects in the DOM are not named (yet). However, they can be identified by an index[1]. This index is passed to the visitor functions to retrieve the CORBA object references.

Generic Approach

To avoid reprogramming the interlink when new sensors or actuators are introduced into Lumina (e.g., a desk light), the following concepts are implemented.

Because all sensors inherit from `CX_sensor`, and all actuators inherit from `CX_actuator` (see 2.1.1), we can cast down the CORBA object reference returned by a visitor function to a reference to `CX_sensor`/`CX_actuator`. The functions `getValue()`/`setValue()` are defined for `CX_sensor`/`CX_actuator`. Therefore, we can call these functions to access our sensors or actuators from the downcasts and do not need to write code for each possible sensor and actuator type.

1. If the DOM is modified, the mapping information might have to be changed accordingly.

```
# starting time:    month    day      hour     minute    second    1/10th of a second)
starting-time       12       21       9        10        1         1
# time step:                          hour     minute    second    1/10th of a second)
time-step                             0        0         0         5

# port number where the control system connects to
cs-server-port                        3000
# IIOP reference of the root object (ORB specific)
corba-initial-reference   localhost   5678     LuminaRoot

# lookup-table entries
# instance-name OMreference Sensor/Actuator Type(name of CORBA ob's w/out CX_)
lookup-data
ils1li1           0/0       S   Receiver # IlluminanceSens[1]
lgt1li1           0         A   Light    # Luminaire
```

Fig. 13: Mapping File (for the example in section 3)

Mapping-File and Mapping-Table

The mapping information is contained in a mapping-file (see fig. 13), which is read by the interlink on start-up. A mapping-table is constructed from this file. The table contains the instance-names of control system objects and CORBA object references of the associated DOM objects.

The `starting-time` and the `time-step` keywords are used to control simulation time. The socket endpoint for the control system prototype is specified after the `cs-server-port` and the location of the CORBA object `CX_root` follows the `corba-initial-reference` keyword (The location is ORB specific; we use ORBacus from Object Oriented Concepts [OOC99]).

The actual mapping information is listed one line per object after the `lookup-data` keyword. A line contains the instance-name (as known in the control system), the Lumina specific object index, the kind of object—sensor (`S`) or actuator (`A`)—, and the type of the object (corresponding to the name of its visitor function).

1. The object types are called Receiver and Light for historic reasons.

2.4.2 Implementation

Java was chosen as an implementation language. Therefore, the interlink can reside on any platform for which the Java Virtual Machine is available. The interlink can be run close to the simulator on a Personal Computer or can be run on a UNIX machine where our control system prototypes usually reside.

Although Java Platform 2 contains an ORB, we chose to use OOC's ORBacus for Java because we are using it on the Lumina side and also because ORBacus allows us to resolve the initial object reference using a fixed location instead of using the NamingService.

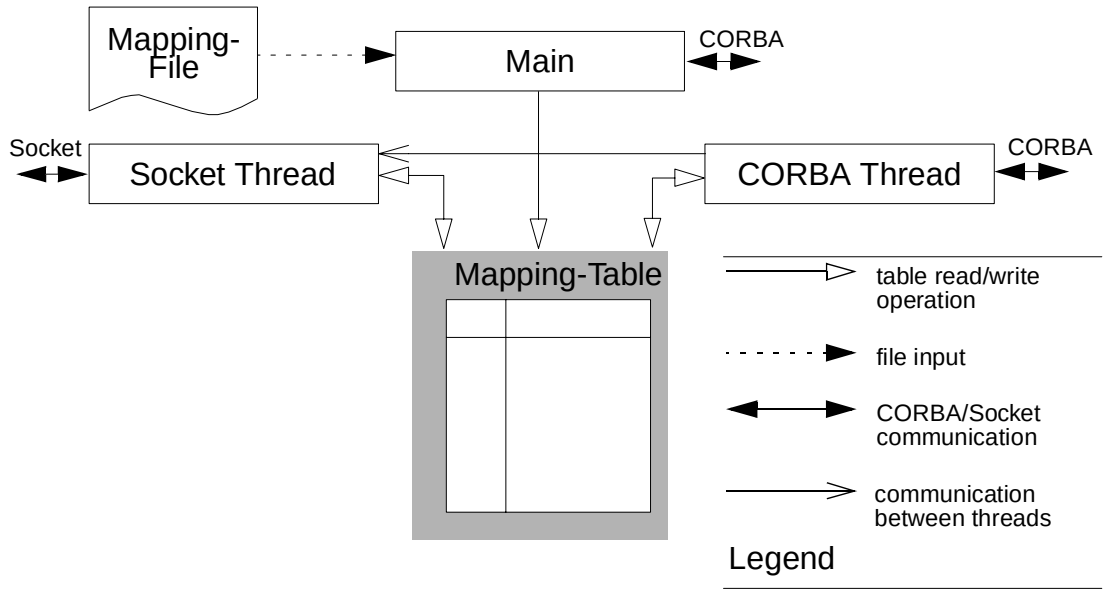

Fig. 14: Thread Structure of Interlink

The CORBA method call that starts a simulation step is blocking, i.e., the execution of the calling program is halted until the method returns. While a time-step is calculated socket messages can be emitted by the control system. The interlink should read these messages and store them. To accomplish this, different threads[1] for each connection endpoint were implemented as shown in fig. 14.

After the main method has read the mapping-file and initialized the mapping-table by retrieving the neccessary CORBA object references, the socket endpoint is created. After this, two threads are invoked that run independently. The first thread handles the complete socket communication, whereas the second thread handles all CORBA activities. Data is shared through the mapping-table.

1. Threads are lightweight processes that execute in the context of the creating process.

When a time-step has been completed, the CORBA thread notifies the socket thread to send the "clk1sys1 <n>" message to the control system. Further, sensor data is stored in the mapping-table, and actuator commands are read from it and sent to the simulator.

If the control system requests sensor information, the socket thread reads from the mapping-table and sends back the desired information. Actuator commands from the control system are stored in the table until a time-step has been calculated. After the actuator commands have been sent to the simulator, they are deleted from the mapping-table.

Section 3 An Example

This section presents a simple example of a lighting control system for which a prototype was generated and connected to Lumina. It employs a simple closed loop control to reach a predefined illuminance in a room using a single luminaire. The room is of rectangular shape and has a window on one side.

Fig. 15 shows the system's object structure. The top level object `Lighting` contains the control algorithm. `IlluminanceSens` represents the illuminance sensor (`CX_receiver` in Lumina). It returns the illuminance inside the room in Lux (lx). `Luminaire` is the actuator object of the lamp (`CX_light` in Lumina). The lamp can be dimmed between light-levels from 0% to 100% in 10% steps.

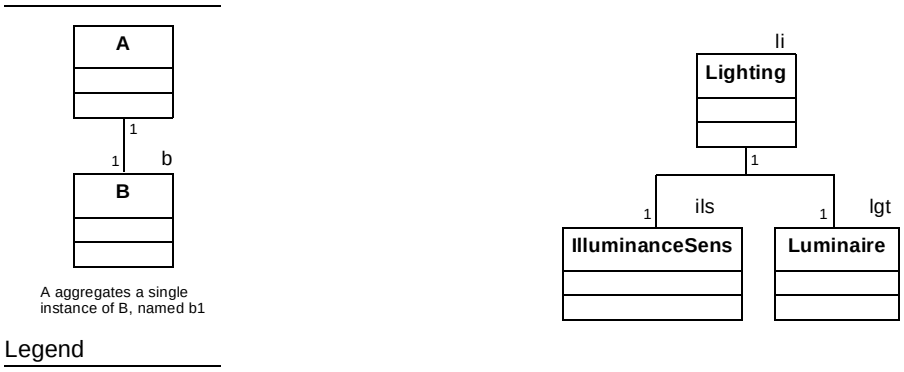

Fig. 15: Object Structure of a Lighting Control System

This object structure is specified in SDL by applying the block type modeling template as shown in fig. 16.

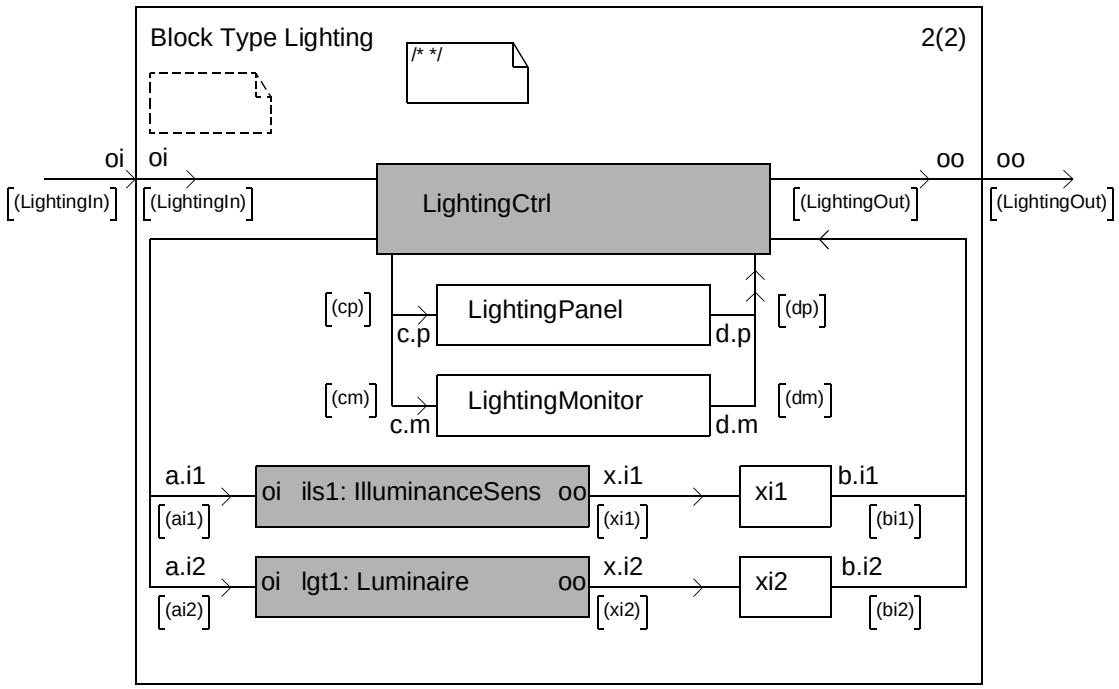

Fig. 16: SDL Block Type of Lighting Control System

The Control Algorithm

The control algorithm determines if the actual illuminance a is above or below the set-point s. To avoid oscillation, the value e is introduced. If $a < (s - e)$, the light-level will be incremented, and if $a > (s + e)$, the light-level will be decremented. Otherwise, the light-level will not be changed. Further, a check if the resulting light level is within valid boundaries (0%–100%) is performed. Fig. 17 shows the specification of this algorithm in SDL's state diagram notation.

The illuminance a is determined by the illuminance sensor object `ils1`, which is mapped to a `CX_receiver` object in Lumina. The luminaire `lgt1` is mapped to a `CX_light` object.

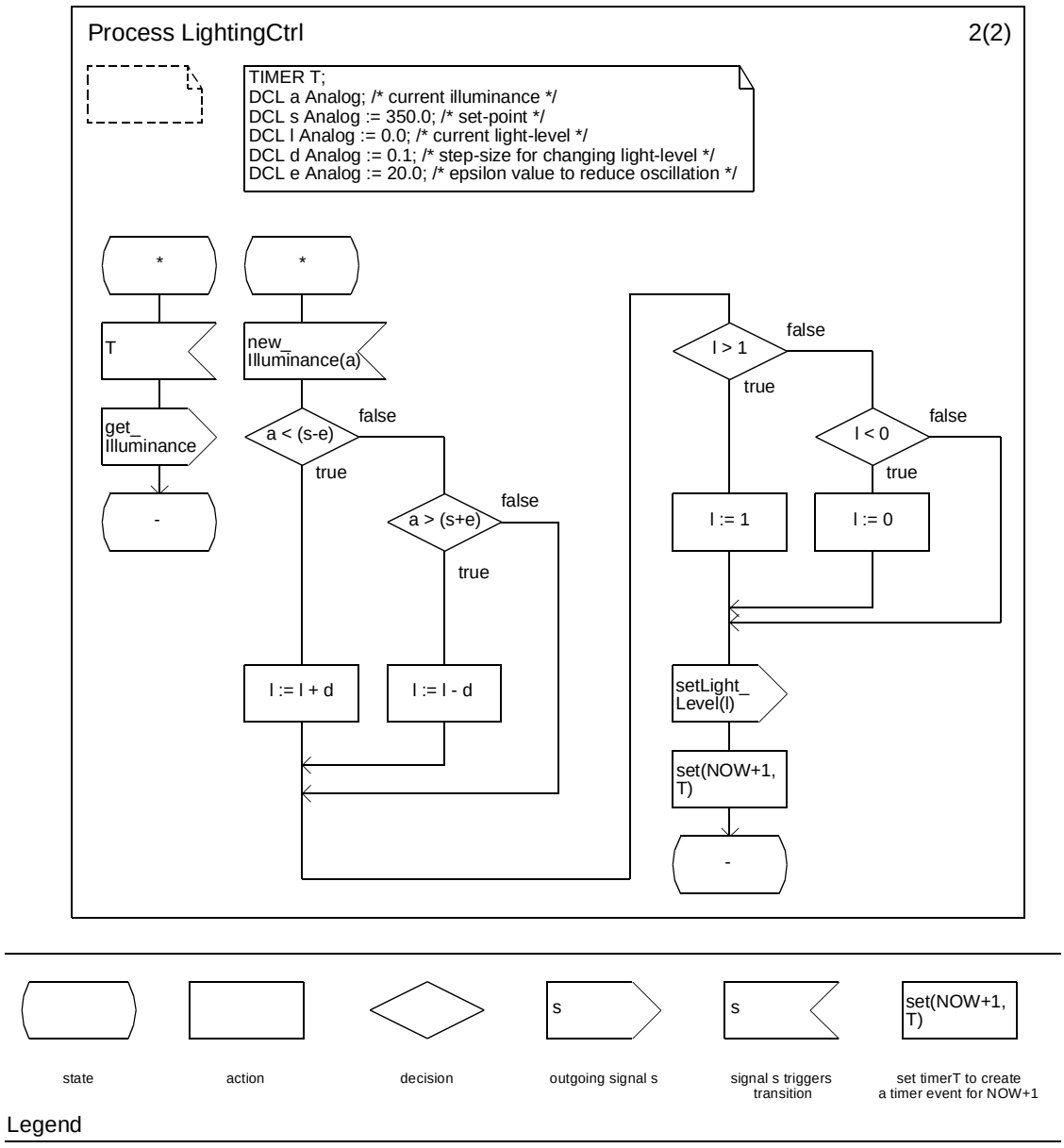

Fig. 17: SDL State Transition Diagram of Lighting Control

Results

Fig. 18 shows the results for our small system. The simulation was performed for 500 minutes of simulated time, starting on 8:30 on April, 1st. The graph shows the illuminance in the room with the lighting control system running compared to the illuminance that were to be measured in the room without the control system. Additionally, the light-levels of the luminaires are shown.

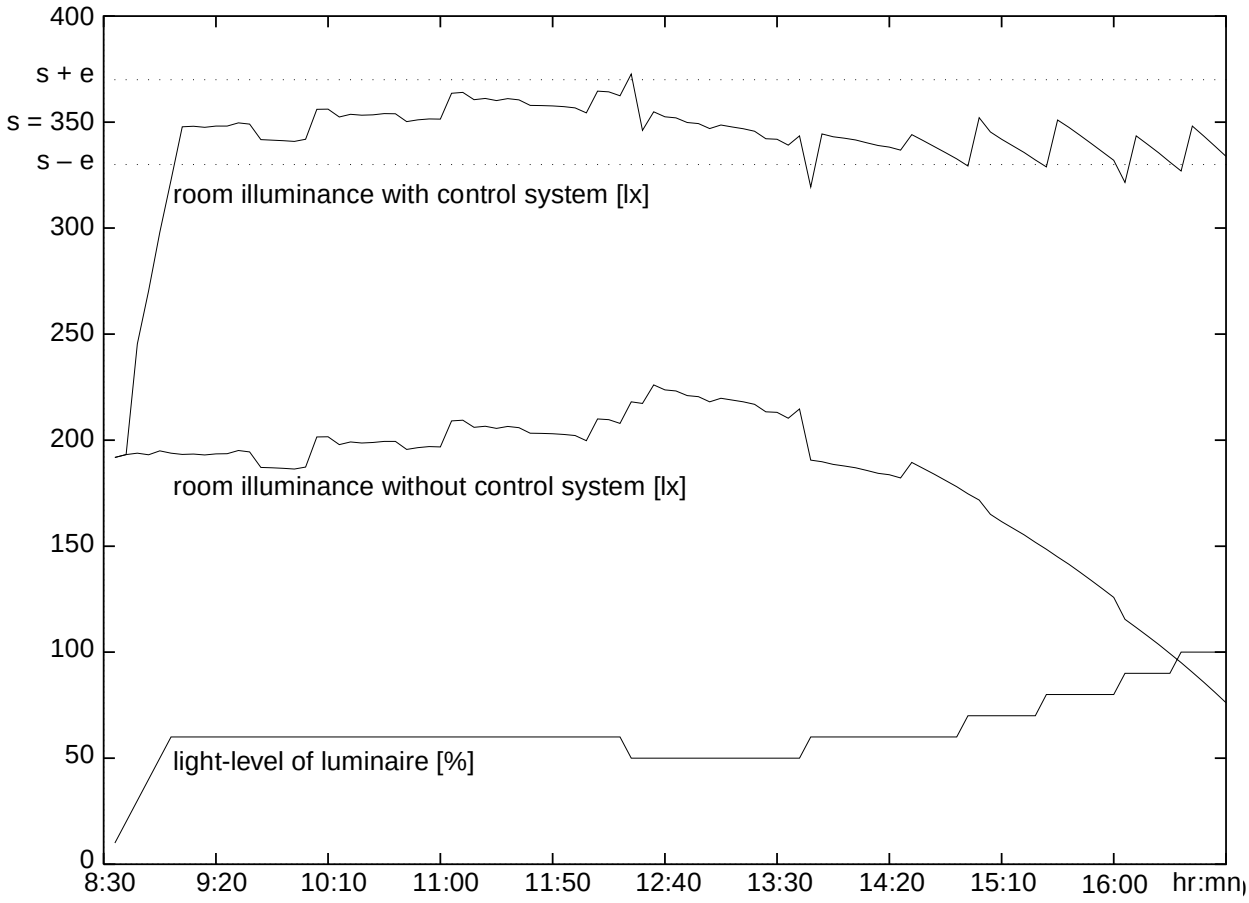

Fig. 18: Simulation Results (April, 1st)

Note, that this simple example wasn't chosen to present an 'intelligent' control algorithm but to prove the synchronization mechanism to be working. A more complex example can be found in [Hal99] where a system is presented that controls four luminaires and one louver in such a way that the desired illuminance is reached with minimal energy consumption.

The example also shows another advantage of simulating the environment. You can evaluate the changes that are caused by introducing a control system. In a real building only the result of the controlled room can be measured for a specific day. It is difficult to determine if two days are identical and can be used for a comparison in real life.

Section 4 Perspectives

Traditional building control systems control the building's **environmental systems** (e.g., heating, lighting, or ventilation) independently. Therefore, an optimal performance might not be achieved. To illustrate, if we try to minimize energy consumption—for each environmental system independently—sunlight is the preferred source of light because it is available at no cost. However, sunlight can heat up the space, thus additional energy will be needed for cooling. Using artificial lighting and blocking part of the sunlight can achieve a better performance [CaD91].

Therefore, **integrated control systems**, which control all of the building's environmental systems, have to be developed [HaL86]. To validate such systems, they have to be connected to building simulators that return results for each environmental system that is controlled by the integrated control system.

Hence, the integration of the control domain into SEMPER should be evaluated. So far, SEMPER's simulation modules utilize local control loops and no integrated control system is implemented. Several problems arise if the control system is added to SEMPER as an Application (refer to fig. 2) representing the control domain. One major concern is the synchronization and communication between the control domain and other domains. The problems that were pointed out when trying to have the control system prototype and Lumina running in a synchronous manner show that this is not trivial. It is complicated by the fact that each simulation module might have interdependencies with the others and with the control system. The interlinking component has to connect the control system prototype with the other applications according to SEMPER's architecture.

Another approach is to connect the control system to SEMPER via SOM-App. The SOM has to be extended by dedicated connection points (i.e., sensors and actuators) that are mapped to objects in the various DOMs. If a CORBA request for an actuator arrives at SOM-App, the

request will be dispatched to the matching object in the DOM. Accordingly, if results for a DOM are available, the sensor objects are updated in the SOM.

Then, the interlink can be used to connect the prototype of the integrated control system with SEMPER. The implemenation of the interlink would be simplified because dependencies between the domains would not have to be considered.

After the control system prototype has been tested using a simulator, testing the system in the physical environment seems neccessary. A simulator uses an abstraction of the physical building. Therefore, simulation results might differ from real world results. To introduce the physical environment iteratively, some parts of the control system can be interlinked with parts of the real building while other parts can still remain connected to the simulator.

Section 5 Conclusion

This paper presented an implementation of an interlinking component to connect building control system prototypes to the lighting simulator Lumina. The interlink was used to convert internet socket messages to CORBA requests and vice versa. A table was used to map control system objects to Lumina objects.

Further, the interlink was employed to synchronize the prototype and the simulator; i.e. the control system prototype was halted while the simulator was calculating results.

Generic approaches like inheritance were applied where suitable. CORBA's Dynamic Invocation Interface was used to avoid the need of reprogramming the interlink if new object types will be introduced.

To show that the interlink is working, a simple control system was specified in SDL, code was generated and a typical scenario was tested.

Acknowledgments

My thanks go to Vivian Loftness, Volker Hartkopf and Ardeshir Mahdavi for giving me the opportunity to perform my research project at the Intelligent Workplace; to Paul Mathew, Kristie Bosko Mertz, Vineeta Pal, Emre Ilal, Sila Berkol and Stephen Lee for aiding me in realizing the project. Finally, I thank Gerhard Zimmermann for sending me to Carnegie Mellon University.

Bibliography

[Aug92] G. Augenbroe. “Integrated Building Performance Evaluation in the Early Design Stages” in *Building and Environment*. 27(2). 1992. pp.149–161.

[AuW91] G. Augenbroe. F. Winkelmann. “Integration of Simulation into the Building Design Process” in J.A. Clarke, J.W. Mitchell, R.C. Van de Perre, eds. *Proceedings Building Simulation ’91 IBPSA Conference*. 1991. pp.367–374

[CaD91] R.A. Carlson, R.A. DiGiandomenico. *Understanding Building Automation Systems*. Kingston, MA: R.S. Means Company, Inc. 1991

[HaL86] V. Hartkopf, V. Loftness. “Integration for Performance” in R.D. Rush, ed. *The Building Systems Integration Handbook*. John Wiley & Sons. 1986

[Hal99] K. Haller. *Development of a Lighting Control System in SDL*. Project Thesis. Computer Science Department. University of Kaiserslautern. 1999

[HeB94] D. Hearn, M.P. Baker. *Computer Graphics*. Second Edition. Englewood Cliffs, NJ: Prentice Hall, Inc. 1994

[Jal97] P. Jalote. *An Integrated Approach to Software Engineering*. 2nd Edition. New York: Springer-Verlag. 1997

[MMB98] A. Mahdavi, P. Mathew, R. Brahme, et al. *The SEMPER Project: A Multi-domain Design and Simulation Environment*. Interim quarterly report to the National Institute of Standards and Technology (9-10-11 ’98). Built Environment Research Laboratory. School of Architecture. Carnegie Mellon University. Pittsburgh, PA. 1998

[Mah96] A. Mahdavi. *A New Computational Environment for Simulation-based Building Design Assistance*. Proceedings of the 1996 International Symposium of

CIB W67 (Energy and Mass Flows in the Life Cycle of Buildings). Vienna, Austria. 1996. pp.467–472.

[Met98] A. Metzger. *Tool Supported Prototyping for the Specification and the Design of Building Automation Systems*. Masters Thesis. Computer Science Department. University of Kaiserslautern. 1998

[MMB99] A. Mahdavi, P. Mathew, R. Brahme, et al. *The SEMPER Project: A Multi-domain Design and Simulation Environment*. Interim quarterly report to the National Institute of Standards and Technology (12/'98-1/'99-2/'99). Built Environment Research Laboratory. School of Architecture. Carnegie Mellon University. Pittsburgh, PA. 1999

[MML96] A. Mahdavi, P. Mathew, S. Lee, et al. *On the Structure and Elements of SEMPER*. Proceedings of the 1996 ACADIA Conference. Tucson, AZ. 1996

[MQS96] A. Metzger, S. Queins, B. Schürmann. *Installation eines Testraums zur Gebäudeautomation und deren Schnittstelle*. Internal document. Computer Science Department. University of Kaiserslautern. 1996

[OFM94] A. Olsen, O. Færgemand, B. Møller-Pedersen, et al. *Systems Engineering Using SDL-92*. Amsterdam: Elsevier Science B.V. 1994

[OMG98] *The Common Object Request Broker: Architecture and Specification*. Revision 2.2. Framingham, MA: Object Management Group (OMG). 1998 http://www.omg.org/corba/corbaiiop.html

[OOC99] *ORBacus*. Billerica, MA: Object Oriented Conecpts, Inc. 1999 http://www.ooc.com/ob/

[PaM99] V. Pal, A. Mahdavi. *A Comprehensive Approach to Modeling and Evaluating the Visual Environment in Buildings*. School of Architecture. Carnegie Mellon University. Pittsburgh, PA. 1999

[Que98] S. Queins. *Modellierungsrichtlinien für den SDL-Entwurf*. Internal Document. SFB 501. Computer Science Department. University of Kaiserslautern. 1998

[Rag93] S.A. Rago. *UNIX System V Network Programming*. New York: Addison-Wesley Publishing Company. 1993

[SDT98a] *(62) The Master Library.* SDT 3.4 Online Help. Telelogic, AB. Sweden. 1998

[SFB99] *SFB 501 - Team3: Baseline Experiment 2 BaX2*. University of Kaiserslautern. 1999 http://www.sfb501.uni-kl.de/team3doc/WWW/BaX2/baX2.html

[Sie98] J. Siegel. "OMG Overview: CORBA and the OMA in Enterprise Computing" in *Communications of the ACM*. 41(10). October. 1998. pp. 37–43

[Tel99] *Telelogic: SDT.* Sweden: Telelogic, AB. 1999 http://www.telelogic.se/solution/tools/sdt.asp

[Z.100] *CCITT Specification and Description Language (SDL)*. Recommendation Z.100. Geneva: International Telecommunication Union (ITU-T). 1993

[Zim98] G. Zimmermann. *A Domain Specific Model Architecture for Complex Embedded Systems: A Building Automation Case Study*. SFB 501 Report 10/98. Computer Science Department. University of Kaiserslautern. 1998