PATTERN-BASED GENERATION OF CUSTOMIZED, FLEXIBLE BUILDING SIMULATORS

JAN PETER RIEGEL, MARTIN SCHÜTZE,
GERHARD ZIMMERMANN
Dep. of Computer Science
University of Kaiserslautern
Germany

This paper describes a domain-specific software development method for the creation of building simulators. The method is based on object-oriented modeling, design patterns and code generation principles. The goal is to provide customizable building simulators that exactly simulate those physical effects an application demands. The numerical accuracy and different algorithms to be used can be tailored to the application's needs. By using object models and preconfigured design patterns, a well-structured simulator model can be created. From this model, the complete product code of a simulator is generated. The patterns help to develop a complete and correct model. Each pattern describes a certain functionality and knows how to generate code to implement this functionality.

1. Introduction

Simulation for performance evaluation of large buildings and their installation is desirable through all phases of the design process. Existing, commercially available building simulators are in most cases large, monolithic systems which neither can be tailored to the simulation needs nor are they easy to use. Worse than that, they frequently require a very detailed description of the building which is not known in early design phases. Therefore, these simulators are used in later stages of the design where the correction of possible errors detected by the simulators is difficult and expensive.

Due to this mismatch between the designer's needs and the simulator's requirements, CAAD system developers try to integrate performance evaluation tools into their systems (e.g., Mahdavi 93). These tools determine important performance indicators during early design phases, sometimes in a two way approach where modifications of the indicators are reflected in possible design modifications (Flemming and Mahdavi 93).

In contrast to the calculation of performance indicators, we propose a 'classical' simulation approach, where buildings are simulated in the time domain. This approach was primarily intended to support building control system engineers with a software test environment, but can also be used for performance evaluation. We do not provide one monolithic, comprehensive simulator. Our software engineering approach to the simula-

tion problem consists of a very flexible simulator generator that creates customized simulators for specific needs and on different levels of abstractions. By this way, the application (CAAD or a control system engineering environment) can be integrated together with the simulator in a design tool (compare Milne 91). The underlying object model of the simulator, as well as the simulated effects, the level of details, and the used physical abstractions can be determined by the user. The modeling of the simulator is supported by a design pattern-based mechanism (Gamma et al. 95/ Pree 95), a concept which originally came from the architectural domain (Alexander, Ishikawa, and Silverstein 77) and found its way to software engineering. When the modeling is done, a code-generator is used to create the product code for an executable simulator.

Chapter 2 describes the simulation principles of our simulators and the different degrees of freedom they allow. The modeling of such a simulator is described in chapter 3. This chapter is divided into three parts, each describing one major step of the modeling. These are: setting up an object model, applying patterns to this model, and transforming a CAD drawing to get a building instance. The generation process is briefly described in chapter 4. The paper concludes with a list of related works (chapter 5) and a discussion of our approach in chapter 6.

1.1 CLASSIFICATION

One application domain where flexible simulators are needed is the development of control systems for buildings. Because it is too expensive and too complicated to test a building control against a real building, simulation is necessary. Here a simulator must act as close as possible like a real building. However, only those effects have to be simulated which influence the or are monitored by the control. Depending on the building control, it might be interesting to test it against a variety of different buildings. The simulator for such control algorithms has to emulate the same timing behavior as a real building, thus real time simulation is necessary.

Another domain where building simulation is useful is Computer-Aided Architectural Design. During the design of a certain building, simulation can be used to verify the performance of construction details. Here, the model of the building is rapidly changing and a simulator must be adapted to simulate exactly the building under development. Simulation should be possible even if only parts of the building are readily designed. Such a simulator is naturally less exact as one using the final design might be, but nevertheless it can be used to detect major design flaws in early phases. By this way, for example, the performance of a central storage heating can be simulated before the entire construction of the building has been completed. If the simulation reveals that the required mass of the heat storage is too big or too small the design can easily be adapted without having to change the complete building.

There are three different categories of simulation environments. *Simulation languages* (e.g. SIMULA, Lamprecht 81) are very universal, but do not provide any help for the development of a domain specific model. To use a simulation language, detailed knowledge of the language and of the problem domain must be present. *Simulation Systems* like GASP (Alan and Pritsker 74) or SMILE[1] are focused on the exact modeling of physical effects. Therefore, they can be used if the physics that should be emulated are well

understood and if useful simulation algorithms are known in advance. Complete *specialized simulators* (e.g. TAS[1]) can be used by people which are not experts in the simulation domain, but tend to be static, monolithic systems that cannot be adapted to the application's needs.

The overall goal of our approach is to provide a powerful, but easy-to-use, method for creating building simulators. The user of this method doesn't need to be an expert in the simulation domain. This way, for example, a tool integrator who knows about the application which has to be tested, and is somewhat familiar with software engineering models but who is not familiar with simulation in detail, can use our method. We try to abstract from the implementation of a building simulator by providing simple and understandable simulation patterns that can be used to model the final simulator.

2. Simulation

The building simulator has to be flexible in several ways:

Different physical effects.   It should be possible to choose from a variety of effects to be simulated. These effects can be isolated or related with other effects (e.g. humidity is related with temperature). In such a case both effects can be simulated in detail or one effect is emphasized while (for simplicity) the other is only approximated. It must be guaranteed that the simultaneous simulation of more effects leads to correct solutions. This is especially true if discrete effects are mixed with continuous ones.

Different levels of abstraction. Depending on which data are actually available and depending on how accurate the simulation should be, a simulator can be more general or can be modeled specific. Therefore, the 'best' simulation algorithm depends on the current status of the application and on the goal of the simulation (i.e. which effect or behavior should be tested).

Accuracy versus speed.   It is a very complex task to simulate physical effects as exact as possible. A customized simulator should be able to trade speed for accuracy in order to get nearly exact results in much shorter time. However, a more precise simulation should also be possible.

Real-time.   The simulation should take place in real time or accelerated according to real time, so that the timing behavior of a building can also be simulated. Furthermore, an asynchronous interaction with the simulator is needed to test control systems.

Hardware-in-the-Loop.   It should be possible to integrate real hardware into the simulator. This restriction comes from the main purpose of our approach, namely to sup-

---

1. Symbolic Interactive Lotos Execution, LOTOS Tool development group, dept. of Computer science, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands
1. TAS (Thermal Analysis Software), Environmental Design Solutions Ltd., 13/14 Cofferidge Close, Stony Stratford, Milton Keynes, MK11 1BY, Great Britain

port a control engineer with a powerful tool to test complete control systems including hardware parts. To integrate hardware in the simulator, the simulation must take place in real-time.

To meet all these needs, an application-specific simulator has to be created. For each physical effect to be simulated, we provide one or more simulation components. The interfaces of these simulation components have to be clear and standardized so that single components can be exchanged in order to include a new effect or to chose a different accuracy. Well-defined interfaces also support the process of stepwise refining a simulator. Starting with a rough, imprecise model of the simulator, it could be refined to incorporate more precise algorithms if special effects are to be simulated. This refinement can be hierachical (i.e. starting with an imprecise simulation of a special effect and using more concrete algorithms in later phases to trade accuracy for speed or to change the level of abstraction) or explorational (adding new effects).

The final simulator is built from several small fragments, each providing a partial functionality. In order to make these parts fit together, the simulation must take place on a very local level. Our simulators consist of many different objects which are closely related to „real world" objects. These objects are, for example, walls or rooms. An *object model* describes these objects and their relations. In contrast to other simulation approaches where the calculation is done on a global level (one big system of differential equations is solved with every time step), our simulation objects simulate themselves making the integration of further functionality easy and leaving room for possible local optimizations. For example, in order to calculate the actual room temperature, a room object collects all the heat-flows from its neighboring objects (walls, heating installations, sun radiation, etc.) and uses these values together with the elapsed time interval to compute the new temperature. The room doesn't have to know, how the heat flow was calculated - this is in the responsibility of the wall or heating objects.

The interfaces of the objects and the data they exchange are declared by predefined design patterns. *Patterns* describe an abstract functionality that can be adapted to objects from the object model. Each pattern of our pattern catalog is able to work together with other patterns enabling to model on different levels of abstraction. This way, one simulation object can easily communicate with other objects. If a variant of an existing simulator is to be created, only few patterns need to be exchanged.

As an example, figure 2 a) describes a very simple situation, where a wall only consists of one object. The heat flow through such a wall can easily be calculated depending on the difference of temperatures of both adjoining rooms (q = (Δv·A)/R_λ, with Δv/[K] = difference of the temperatures, A/[m^2] = area of the wall, R_λ/[m^{2}K/W] = thermal resistance of the wall). In figure 2 b) a wall is more complex. Here, it consists of several layers (i.e. the wall object has associated layer objects), each with its own thermal resistance or heat storage capacity. If such a wall is asked about its heat flow, it delegates the question to its layers which sum up the total heat flow. In either case, a room object doesn't have to know about the wall structure - it just uses the well-defined interface to the wall object.
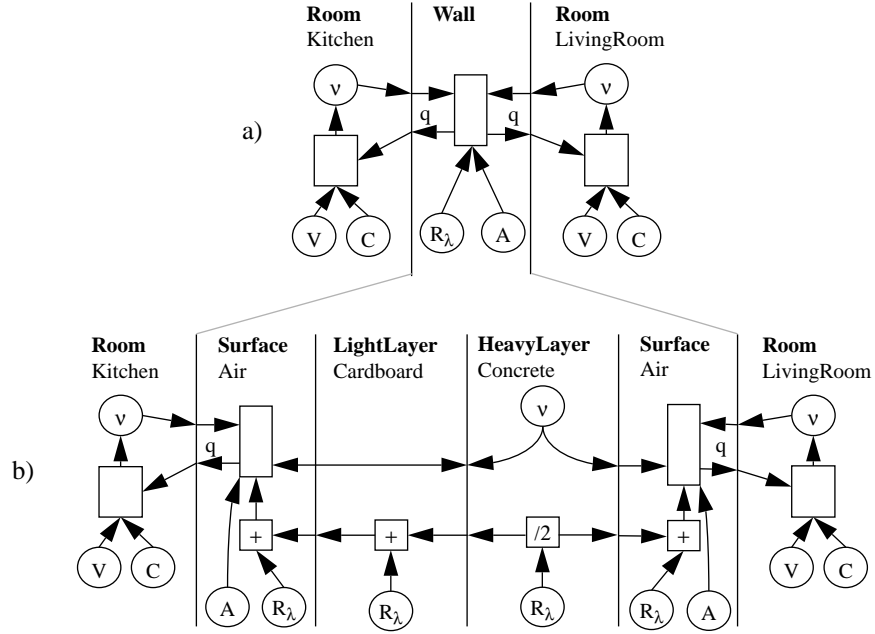
Figure 1. Interfaces of a wall object.
a) simple wall
b) layered wall

3. Modeling the Simulator

The question that now arises is how to specify the simulator without having to deal with the issues of the last chapter in detail.

A software system can be divided into several views. The *static view* describes all parts that do not change while the program runs. Especially data structures belong to the static view. These can be modeled using object diagrams (Rumbaugh et al. 91) or Entity Relationship diagrams. There are several methods how to model data structures. Tools are available to support the modeling and generation of program code out of these models. Static models can be used as an integration platform for the complete system: in these models references to other models describing the functionality or the behavior of a certain object can be stored.

Another aspect of the static view is the instantiation of the object model. That is, the actual data on which the program operates have to be described. We use object models to describe static aspects (see figure 2) and transform data from a CAD editor to instantiate these models, thus building the input to our simulator.

The *dynamic view* on a software system shows how the program behaves during run time. To create dynamic models (e.g. State Transition Diagrams), a software-engineer has to know very much about the problem domain and needs experience in software

design. For simplicity, we "hid" all dynamics in special control patterns and in a simulator kernel library. This library provides methods to control the simulation process and can be used on the modeling level. Using control patterns, the dynamic behavior of a simulator can be modeled with abstract descriptions instead of setting up a detailed State Transition Diagram from scratch.

Finally, the *functional view* on a software system has to be described. This is normally the domain of a programmer or software engineer who has to choose the best fitting algorithms and translates them into program code. Because of the limited application domain, we are able to encapsulate possible algorithms in simulation patterns. A pattern generator knows how to implement the pattern's functionality so that the user can concentrate on the functionality a pattern provides and doesn't have to bother about a correct implementation.

The customization of the simulator takes place on the modeling level. The central model is an object model describing the building's topology (see a very simplified example in figure 2). This model may be adapted by the user to describe the building as exact as
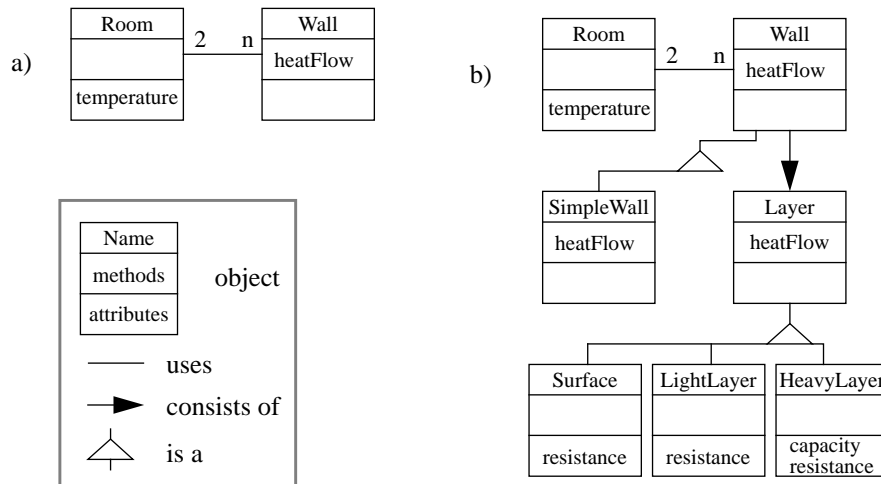

Figure 2. Object models for different simulation methods.
a) direct calculation
b) delegation

needed. For example, the model can be enriched with additional elements describing exact geometry or detailed information about installations and control systems. Figure 2 a) is a small model sufficient to represent simple wall structures (compare figure 1 a)). With a building model like in figure 2 b) more sophisticated simulation approaches as in figure 1 b) can be modeled. In addition to this building model, there are other models describing the simulator kernel objects and the simulator's functionality. To create a special-purpose simulator, these different models have to be linked together. As a 'glue' between the models we use a derivative of design patterns (Gamma et al. 95).

Figure 3 describes the development of a building simulator (see Altmeyer et al. 97). The central model is an object model describing the static view on the simulator. Here,

objects like 'Room' or 'Wall' are declared. To describe the behavior of these objects, we use simulation patterns from our pattern catalog. These patterns are 'bound' to the object model by stating how they interact with the objects. A set of generators and transformers is used to generate code from the resulting model.
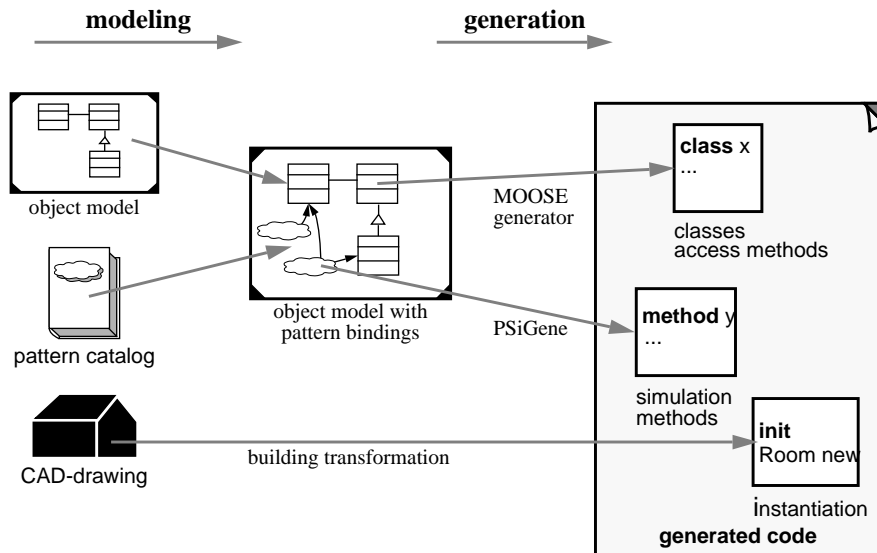

Figure 3. Simulator development with MOOSE/PSiGene

Starting with the adaptation of the building model (i.e. an object model describing a building), design patterns are successively applied to the model in order to get the needed simulator functionality. The result of this modeling step is a refined building model which includes patterns and their bindings to the objects from the model. Encapsulated in each pattern is a code template so that after the modeling step a complete simulator can be generated. The class structure of the simulator program is generated by our software engineering tool MOOSE (Model-based Object-Oriented Software Generation Environment, see Altmeyer, Schürmann, and Schütze 95), whereas the functionality is created using the PSiGene generator (Pattern-based Simulator Generator, Schulz 97). The concrete building instance, which should be simulated, is acquired using a CAD-editor (Speedikon[1]) and transforming the drawing into usable program code. In the following, the three major parts of the modeling process are explained: adaptation of the static models, binding of design patterns, and instantiation of the building. The generation process is briefly explained in chapter 4.

3.1 OBJECT MODELS

A central point of the simulator model is the structural view of the simulation objects. Here, the objects are described along with their attributes and relations to other objects.

---

1. Speedikon X, IEZ AG, Berliner Ring 89, D-64625 Bensheim, Germany

The notation for this description is a derivative of the OMT class models (see Rumbaugh et al. 91 and figure 2).

These models can be edited using MOOSE, a software engineering tool we have written. MOOSE also contains powerful software generators which are capable of generating complete class structures out of these object models including access functions, methods to handle connections between objects, and a persistent storage mechanism. These code generators are available for the programming languages C, C++, and Smalltalk (Visual-Works). For our building simulator, we use the Smalltalk generator.

The modeling of object models goes hand in hand with the application of patterns to these models (see below). We provide a basic building reference model (i.e. an object model describing the static view on a typical building) that can be used as is or can be taken as a basis for further refinements.

3.2 DESIGN PATTERNS

Design patterns are used in our approach to define the simulator functionality and act as 'glue' to define the interactions of objects in the object model. They originally came from the architectural domain. The architect Christopher Alexander used patterns to describe what 'good' designs are and how to obtain them (Alexander, Ishikawa, and Silverstein 77). Therefore, he created a pattern language in the way that a set of interacting patterns describes how and why a building is constructed the way it is. He defines patterns as "a three-part rule, which expresses a relation between a certain context, a problem, and a solution". The idea of patterns has been transformed to the domain of Software Engineering (Gamma et al. 95). Here, each pattern describes a recurring software problem which should be solved, a context in which this problem occurs, and a solution to this problem.

In general, software design patterns give clues how certain problems can be solved using special object structures or algorithms. Design patterns are mainly used to support the modeling and the documentation of software. They should be applicable in many situations, therefore, they are on a rather abstract level and not useful for automatic code generation. In our case, however, we focus on a narrow domain, the domain of building simulation. Therefore, we were able to create specialized patterns that describe small parts of a building simulator. These specialized patterns contain code fragments so that an automatic code generation is possible if such a pattern is used (i.e. bound to an object model).

Each pattern contains a description *when* it could be used, *which* problem it solves, *how* to apply it, *which* underlying object structure it expects, and *how* a generator should generate code to solve the problem. A pattern contains only the solution to a small problem. To solve a complex problem, it often depends on functionality provided by other patterns. For example, to simulate the temperature of a room, the room object needs to know how to calculate the temperature out of incoming heat flows. A pattern *Thermal-Mass* is responsible for the calculation of the temperature and relies for the calculation of the heat flows on functionality provided by other patterns. The parts of a pattern which are defined elsewhere are called 'hot spots' (Pree 95) because the pattern itself only defines the interface making different implementations (by other patterns) possible. Thus

6.1. Pattern ThermalMass

Intent

The ThermalMass pattern computes the temperature of a mass depending on the amount of heat affecting the mass. …

Also Known As
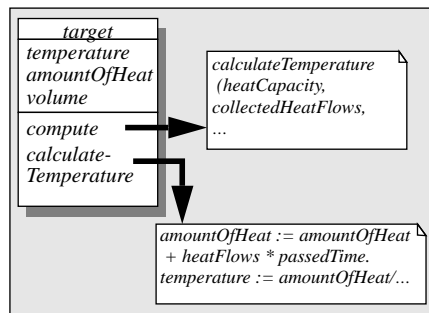
Simulation thermischer Masse [1]

Motivation

A volume has to act as a thermal mass to compute its temperature. …

Applicability

This pattern can be bound to any thermal mass. Typical this is a room or …

Structure

Participants

Objects
- target
  Object to bind pattern to.

Attributes
- temperature
  Last computed temperature.
- volume
  Volume of the thermal mass.

Methods
- compute
  Does the calculation cycle for one time.

Interfaces
- getHeatCapacity
  Determines the storage capacity from the temperature of a mass.
- collectHeatFlows
  Determines the heat-flows from and to a thermal mass.

Collaborations

The computation relies on …

Consequences

…

Implementation

Smalltalk-Code-Templates
- *{init}*
  self *{amountOfHeat}*:= (self *{temperature}* *
  self *{volume}* * self *{getHeatCapacity}*).
  self *{timeOfLastComputation}*:
  Scheduler simSched simMillisecondClockValue.
- *{compute}*
  | heatCapacity collectedHeatFlows timeNow |
  timeNow:= Scheduler simSched simMillisecond-
  ClockValue.
  heatCapacity := self *{getHeatCapacity}*.
  collectedHeatFlows := self *{collectHeatFlows}*.
  self *{calculateTemperature}*WithCapacity: heatCapacity
  withHeatFlows: collectedHeatFlows while:
  ((timeNow-self*{timeOfLastComputation}*)/1000).
  self {*timeOfLastComputation*}: timeNow.
  ...

Related Patterns

Thermal Exchange

Figure 4. Pattern example (abbreviated)

each pattern includes a description of the interaction with other patterns by using **template methods** which include interface and implementation of a certain functionality and **hook methods** which only describe the interface.

Figure 4 shows parts of one of our patterns (ThermalMass) in detail. Every pattern is structured into several sections to describe the context, the problem, and the solution separately. 'Intent' and 'Motivation' describe the addressed problem using a suitable example from the application domain. 'Applicability', 'Participants', and 'Collaborations' show the context and define how a pattern can be used. 'Structure' and 'Implementation' show the solution. The organization of a pattern is the same as in (Gamma et al. 95), but we have formalized some parts to make code generation possible. The 'Participants' part describes the exact interface of a pattern, i.e., every parameter of the pattern that has to be bound to the object model. In our example, the pattern needs a binding to a target

class (e.g. room). This object has to have two attributes *temperature* and *volume*. Also a name for the template method *compute* has to be specified. To calculate a room's temperature, the heat flow into it has to be known. Because there could be many different heat sources and different calculations of the heat flows, the method *collectHeatFlows* is only defined as a hook method. The actual implementation has to be done elsewhere (i.e. with another pattern bound to the wall or to the heating installation). Last not least, the heat storage capacity of the room must be specified. This physical value is depending on the actual room temperature and on the material with which the room is filled (i.e. air). Therefore, this pattern defines only the interface to a method *getHeatCapacity*. With the specification of these six parameters, the pattern is fully bound to the object model and code can be generated.

So far we have collected 14 patterns in a pattern catalog. This small number of patterns is sufficient to model and create building simulators for thermal effects with many variants. We're now extending our catalog to include more physical effects (humidity, light, exchange of air, and others) and different accuracies for these effects. The pattern catalog is divided into three parts concerning physical effects, simulation artifacts, and structural adaptation patterns. Inside these categories, the patterns are ordered using aggregation and inheritance, making it easier to find special patterns.

Each group of patterns that provides a similar functionality uses the same interface. Therefore, it is easy to exchange some patterns in order to build a simulator variant; only local changes are necessary, the rest of the simulator model doesn't have to be changed.

To use a pattern, every interface element which is described in the 'Participants' section must be bound to the object model. Optional bindings are marked in the pattern description. The binding is currently done using a textual description language. We are implementing a graphical editor to be able to perform pattern bindings more easily. Each participant can be bound to an object, an attribute, a method, or to a valid Smalltalk statement, depending on its type.

For example, to simulate the temperature of a room, a 'room' object is needed. The actual temperature depends on heat flows that flow from or to the room through walls or heating equipment. So the room has to be related with its surrounding walls. Therefore there also must be a wall object which is connected to the room. To simulate the temperature of a room, a pattern ThermalMass can be bound to it. With this pattern the room is able to update its temperature using incoming heat flows. The heat flow through a wall depends mainly on the difference of temperatures of its adjoining rooms. A pattern ThermalJunction provides the functionality to calculate these heat flows. Furthermore, the calculation of the room's temperature has to be continuously stimulated so that it is always up to date. The pattern ContinuousComputation binds the room to our simulation kernel library providing just this functionality (see figure 5). Mainly with this three pattern bindings a first, very simple simulator can be modeled.

The binding for the pattern ThermalMass (figure 4) to an object 'Room' is textually described with the following statements:

```
ThermalMass                                        // Name of the pattern
    bind: 'target' to: 'Room';                     // Target class
    bind: 'temperature' to: 'temperature';         // Bind attribute 'temperature'
    bind: 'volume' to: 'volume';                   // Bind attribute 'volume'
    bind: 'compute' to: 'calculateNewTemperature'; // Rename template method calculate
    bind: 'getHeatCapacity' to: 'getHeatCapacity'; // Hook method getHeatCapacity
    bind: 'collectHeatFlows' to: 'collectHeatFlows'. // Bind name to hook method
```

Figure 5 shows some sample bindings graphically. Patterns are not only used to provide simulation functionality to one object, they also act as 'glue' between different objects.
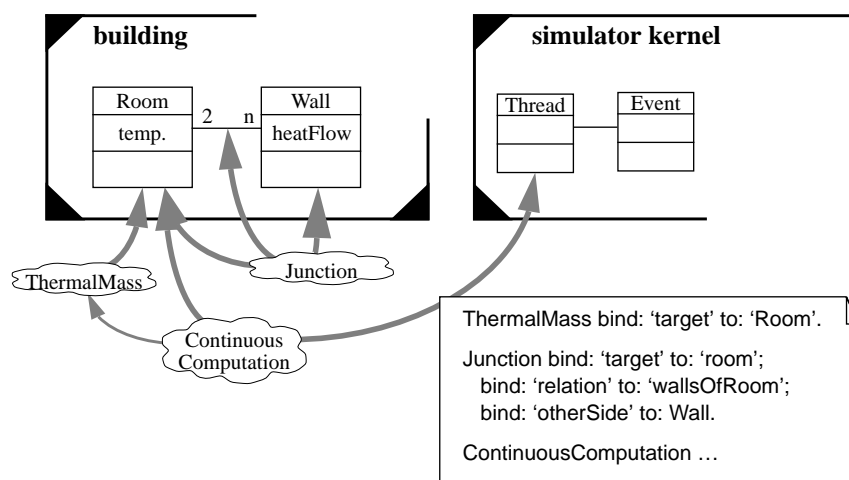

Figure 5. Pattern bindings (sample)

For example, the pattern 'ContinuousComputation' is bound to the Room and to an object 'Thread' from the simulator kernel. This means, that the temperature of a Room should be continuously simulated using a Thread that is controlled by the simulator kernel, 'gluing' a Room with a Thread.

3.3 BUILDING TRANSFORMATION

After the simulator is modeled and generated, it can be used to simulate any building that can be expressed by the building model. But to simulate one special building, the model has to be instantiated. We use a conventional CAD program to draw the plan of the building to be simulated and transform it into code that can be used directly by the simulator. This transformation is done semi-automatic.

We have chosen Speedikon as input to the transformation, because it is object-oriented and operates with objects like 'walls' and 'rooms'. Therefore the transformation to our building model is easy.

Speedikon doesn't handle physical descriptions of construction details like, for example, heat storage capacities of walls. These values have to be entered manually. We use heu-

ristics to get a useful building instance that can be simulated (e.g. by assuming default values for material constants). Manual adaptations are needed to mirror the exact physical constants of the building. By now, the transformation is hard coded for one building model but we will try to use transformation rules in order to be able to handle a greater variety of building models.

4. Generating Simulators

After modeling the building simulator, program code has to be generated to get an executable generator. We have written a code generator called PSiGene that creates Smalltalk code from the patterns' code templates. With different code templates, PSiGene is also capable of generating code for other programming languages. The generation takes place in a two phase process.

Each pattern from our catalog has been coded as a Smalltalk class. During the first generation phase every pattern which was used in the model becomes instantiated. Afterwards PSiGene performs syntactical checks. For example, every mandatory pattern binding must be made, and for every hook method of a pattern a template method has to be defined somewhere. At the end of the first phase, a detailed report of the pattern instances is created.

All the code generation is done in the second phase. In a simple case, code generation can be as easy as copying the pattern's code templates to the target classes while performing simple macro replacements. However, the generator has enough knowledge to do some code optimizations. For example, if more than one code template is given, PSiGene can choose one that fits best to the building model. This is due to the fact that PSiGene does not only know of the pattern bindings, but also regards the building model. Some of our patterns (like StateMachine) use software synthesis techniques because their functionality cannot be fully described using simple code templates.

Figure 6 shows an example of the running simulator. The graphical in- and output consists of several components which were manually adapted to the simulator. However, the whole simulation functionality is generated. With special patterns it will be possible to model and to generate the user interface, too.

So far, we have modeled several small building simulators using patterns. An example we created, is a typical simulator for thermal effects consisting of 24 patterns (11 different types of patterns were used). We suppose that each extra physical effect to be simulated takes about 5 additional types of patterns. The number of used patterns depends on the size of the object model. If many objects should be simulated, more patterns have to be bound to these objects. The object model in this example consists of 15 building object classes and 6 classes from the simulator kernel library. About 5300 lines of code were generated from the building model and the patterns. A building with 9 rooms and several doors and windows consists of about 300 objects.

Comparing the time used to model and generate a simulator using PSiGene with a manually written simulator shows that it is much more efficient using our method even if some patterns which are missing in out catalog have to be created. Building variants of a simulator is a matter of hours (from the concept to the running simulator).
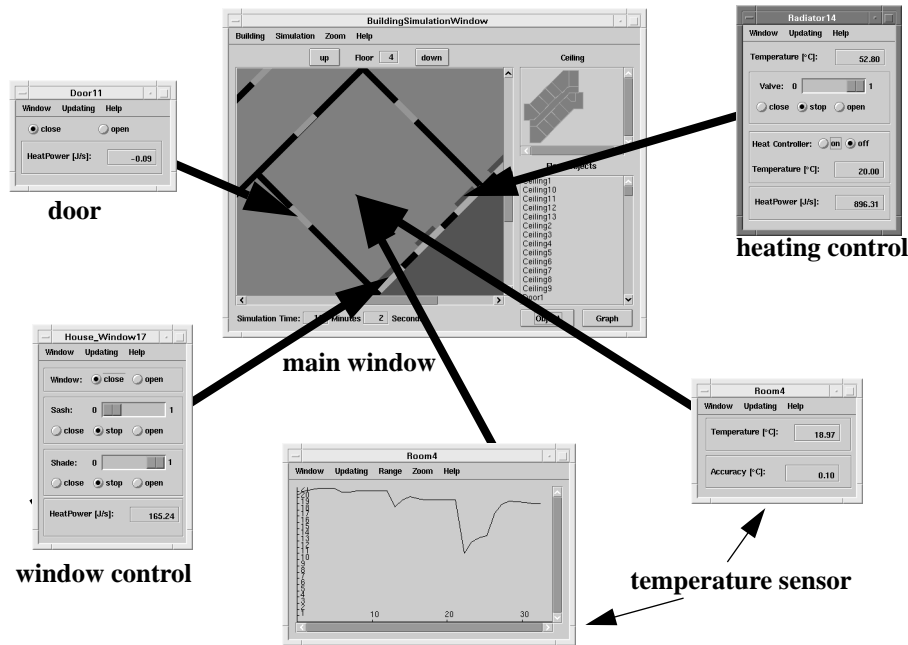
Figure 6. The running simulator

5. Related works

We are providing a method to model and generate building simulators. To do so, we have combined software engineering methods with application specific knowledge from the simulation domain. The use of design patterns to create software models is a very actual and much discussed topic. The main usage of patterns is during the analysis and design phase (Gamma et al. 95, Pree 95), however, there are some approaches using patterns for software generation (Budinsky et al. 96). Our patterns are small, domain specific pieces that can only be used for software design. The main advantage in this is that we can generate the complete product code. We combined software engineering techniques with domain-specific knowledge and software generators. Related work has been done, for example, by Batory (Batory et al. 94).

The modeling of building simulators is a broad research topic with many facets. Using a simulator during the evolution of a building design is done at the Carnegie Mellon University (Flemming and Mahdavi 93, 95). Here, the simulator is used in a two-way approach: the simulation indicates performance issues of a building which could be used in a reverse engineering step to automatically adapt the building to yield a better performance. This simulator is optimized for this reverse engineering step. Explicit modeling of building simulators has also been done by Filiz Ozel (Ozel 91) who uses object diagrams in conjunction with rules to create a simulator.

6. Conclusion and future works

The main advantage of our approach is that we provide a simulator generator instead of a fixed simulator. The user is therefore capable of creating simulators which exactly match his simulation needs without unnecessary overhead and without providing too detailed information.

We are now collecting more patterns to be able to simulate more effects on different levels of abstractions. All our patterns are collected in a pattern catalog which itself is part of a dictionary describing the domain of building automation. This dictionary includes formulas, objects, and models describing different aspects of the domain. We are developing search mechanisms to extract solutions to given design problems from this lexicon. A solution to the problem of thermal building simulation of a given room can be an object model describing room objects and a set of patterns that could be bound to that model.

Further work has to be done to support a user in developing building models using our method. We are trying to incorporate modeling guidelines and checks for correct pattern bindings in a pattern editor we are currently implementing.

From our experience, we believe that this method increases productivity and helps non-experts of the domain to develop useful tools.

References

Alan, A., Pritsker, B. (1974) The GASP IV simulation language, Wiley, New York

Alexander, C., Ishikawa, S., and Silverstein, M. (1977) A Pattern Language, Oxford Univ. Press, New York

Altmeyer J., Riegel J. P., Schürmann B., Schütze M., Zimmermann G. (1997) Application of a Generator-Based Software Development Method Supporting Model Reuse, in Proc. 9th Conference on Advanced Information Systems Engineering (CAiSE*97), Barcelona

Altmeyer, J., Schürmann, B., and Schütze, M. (1995) Generating ECAD Framework Code from Abstract Models, Proceedings of the Design Automation Conference '95, San Francisco, California

Batory D., Singhal V., Thomas J., Dasari S., Geraci B., Sirkin M. (1994) The GenVoca Model of Software-System Generators, IEEE Software, September 94

Budinsky F. J., Finnie M. A., Vlissides J. M., Yu P. S. (1996) Automatic code generation from design patterns, IBM Systems Journal, Vol. 35, No. 2, (http://www.almaden.ibm.com/journal/sj/budin/budinsky.html)

Flemming, U., Mahdavi, A. (1993) Simultaneous Form Generation and Performance Evaluation: A Two-Way Inference Approach, CAAD Futures '93, Elsevier Science Publishers Ltd., Amsterdam, 161-173

Flemming, U., Woodbury, R. (1995) Journal of Architectural Engineering, Vol. 1, No. 4, 147-152

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) Design Patterns, Addison-Wesley

Lamprecht, G. (1981) Introduction to SIMULA 67, Vieweg

Mahdavi, A. (1993) Open Simulation Environments: A Preference-Based Approach, CAAD Futures '93, Elsevier Science Publishers Ltd., Amsterdam, 195-214

Milne, M. (1991) Design Tools: Future Design Environments for Visualizing Building Performance, CAAD Futures '91, Vieweg Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 485-496

Ozel, F. (1991) An Intelligent Simulation Approach in Simulating Dynamic Processes in Architectural Environments, CAAD futures 1991, Vieweg Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 177-190

Pree, W. (1995) Design Patterns for Object-Oriented Software Development, ACM Press, Addison-Wesley

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen,W. (1991) Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, N.J.

Schulz, S. (1997) PSiGene - A Pattern-Based Simulator Generator, diploma thesis, University of Kaiserslautern