A Pattern-Based Application Generator for Building Simulation

Martin Schuetze, Jan Peter Riegel, Gerhard Zimmermann
Computer Science Department
University of Kaiserslautern, Germany

Abstract:
This paper describes a domain-specific software development method based on object-oriented modeling, design patterns, and code generation principles. The example domain is building simulation, however, the approach is general and may be applied to other domains as well. Patterns are used to describe how the simulation objects interact. Code-templates associated with every pattern are used to generate the final application code. The method can be applied to generate large families of customized application frameworks from variations of the models. This is particularly useful for domains where applications have to exist in individually tailored versions for every project.

1. Introduction
In this paper, we describe a domain-specific software development method based on code generation from object-oriented application models. The static aspects of applications are defined in terms of object types, relations, and attributes, as it can be found in most CASE methods (e.g. the Booch Method /Boo91/ or OMT /RBP91/). The techniques to generate code for the static aspects of applications are well-known and will not be described here (see for example /ARS97/). The way the objects interact, as well as their domain-specific operations, are defined with design patterns which provide the 'glue' between objects or even between partial models. Code for operations and interaction of the application's objects is then generated from the glue. Therefore, our method combines techniques for code generation as used in application generators with the design pattern approach /GHJ95/ (see also section 5). In the following, we will focus on the partial formalization of design patterns as well as on the code generation phase, both implemented in a system called PSiGene (Pattern-Based Simulator Generator).

Application Domain
PSiGene was developed within the MOOSE\(^1\) project, mainly to support building control system designers in testing their applications by providing a highly customized, real-time building simulation framework. It is necessary to trade accuracy for speed when large buildings are simulated. At the same time, the simulation should be carried out at different levels of abstraction, incorporating different physical effects. Last but not least, the model of the simulator should be defined by the system designer (which is an software engineer), not by simulation experts. The simulation objects should be

---

1. Model Based Object-Oriented Software Generation Environment, see /ASS95, ARS97/:
MOOSE is a framework for model-driven code generation. It consists of a central database for different types of models, a set of model editors, and several code generators for different software components.
identical to ‘real world’ building objects and control artifacts. It seemed impossible to support all requirements optimally with one fixed simulation library. Therefore, we have chosen a generator approach to create customized and optimized building simulators from object-oriented models and from design patterns defining object interaction and domain specific operations.

Our method is not bound to any particular domain, while instantiations of the method (i.e. instantiations of model editors and generators) are domain specific.

Goals
PSiGene was developed with the following goals in mind:

- Easy integration of domain-specific functions with object-oriented models:
  Modeling of the simulator should take place on the object type level, simulation methods and operations should be encapsulated as far as possible.

- Clear distinction between ‘objects’ and ‘operations’:
  Because the class model of the simulation objects may change from application to application, it is useful to divide the simulation objects from the simulation methods and link both together using domain specific patterns.

- Generation of large portions of code for a specific application domain:
  For a limited domain, we are able to set up powerful models with strict semantics. In our case, we deal with models and well-defined simulation methods for certain aspects of building physics. For these models, it is possible to generate large portions of the code automatically.

- Support of reuse on a medium granularity scale:
  While our overall method ensures reuse of complete application designs as well as code reuse, the patterns should represent what /GHJ93/ calls reusable micro-architectures. The reuse takes place every time a pattern is connected to appropriate objects.

- A flexible way to create customized frameworks:
  The output of PSiGene is a highly application specific simulation framework, optimized for one simulation purpose. By this example, our approach shows that reuse of large software components, the frameworks, can effectively be supported by code generators in conjunction with pattern-based models.

The rest of this paper is structured as follows: section 2 discusses object models, patterns, and frameworks. In section 3, we show how we define patterns for PSiGene. Section 4 describes the generator, and section 5 discusses related approaches. The paper concludes with a discussion of the benefits and limitations of our approach in section 6.

2. Modeling with Patterns
Within our MOOSE system, an application is defined as a set of interacting components. Consequently, the model of the application also consists of several component models, each describing one aspect of the application’s structure and behavior. The component models and their elements are connected by what we call the ‘glue’ of the application.

For PSiGene, we support the following component modeling notations: class models are used to define the structure of the application. This notation is quite similar to those
used by other OOD techniques. At the same time, it is used to express libraries on the modeling level, for example a real-time simulation kernel. The set of initial objects is defined by an object model. We use an architectural CAD editor to provide simulation objects. A finite state machine model is used to express dynamic behavior other than hard coded in the pattern's code templates.

The glue is provided by model aggregation, by ordinary relations, by aggregation relations, by inheritance, and by a set of design patterns as described in the next section. The design patterns define the interaction between objects and encapsulate fragments of the simulator's functionality. They also contain code templates for the generation of simulation methods.

The use of patterns is closely related to the use of frameworks and the reuse of design provided by frameworks. Most authors/GHJ95, Pre95/ emphasize the documentation aspect of patterns: patterns help to adapt frameworks (and of course to create 'good' designs) by stating how the objects of a framework can be customized and integrated, and by describing the interaction between those objects. PSiGene produces a building simulation framework which may be used stand-alone or in other applications such as smart building control software. By using patterns, the coupling and interaction between simulation objects (and between kernel and simulation objects) can be defined on the modeling level while the implementation details are left open to the generator (e.g. linking a thread to a simulation object by delegation or by inheritance). At the same time, the usage of patterns as a glue between the simulation objects allows us to define what quality of service is needed for the current simulation problem, simply by instantiating the appropriate patterns. The generator may then, based on the class model and the patterns, choose from a variety of code templates for the implementation. This leads to the generation of powerful, optimized, and customized frameworks which exactly fulfill the application's needs.

The process of generating a simulator with MOOSE/PSiGene is shown in figure 1. The user specifies the class model of the application. At this time, necessary libraries will already be available as MOOSE class models. The ADT (abstract data type) generator of MOOSE automatically creates a complete class hierarchy for the class model with all create/destroy/access methods.

After setting up the class model, the simulation problem is specified by applying the appropriate, predefined simulation patterns: the patterns are bound to elements of the model (see also figure 3). Due to the descriptive nature of the patterns, the user may find that some classes, attributes, or relations are missing. This leads to a cycle of alternatingly modifying the object model and applying patterns. After finishing this process, PSiGene will link simulation methods to the classes.

The set of initial objects is specified with a standard architectural CAD editor. After transforming the CAD drawing into corresponding objects defined by the class model, these objects can be used by PSiGene to optimize the simulation methods and by the simulation framework to form a complete executable.

Our method can also be applied to related domains, like building control, if the appropriate patterns are provided.
3. Catalog of Patterns

The catalog of patterns we use within PSiGene as well as their textual description is very much influenced by the work of the “gang of four” /GHJ95/ while the graphical notation and the binding was inspired by the work of Pree /Pre95/. But in contrast to these two approaches, we had to further formalize the pattern description in order to allow code generation from instantiated patterns. This means that we had to define the interface of the patterns formally as well as the code templates from which the source code of the application is constructed. We set up a pattern catalog where every pattern is structured as shown in figure 2. Every catalog entry defines the context where the pattern can be used, as well as the problems addressed and their solution. It has a Name and is classified by an Intent. Many parts like Motivation and Applicability are self-explanatory, however, they are written with the application domain in mind so that the information here are very concrete and closely related to the function/behavior associated with the pattern. The Structure is defined graphically, so that the user has a direct feeling for the context the pattern may be used in.

The patterns are bound to the application’s class model by binding formal pattern parameters to objects, attributes, relations, and methods of the class model. The Participants part defines these formal parameters. In contrast to other pattern catalogs, this interface is described by name:type pairs. The name is used internally by a pattern to reference an element, the type defines types of elements that can be bound to the name. Examples are target:Object or volume:Attribute. Currently, PSiGene supports the following types: Object, Attribute, Relation, Method, and Expression.

The Implementation part contains, for every target language, code templates for the pattern which are used by the generator (see section 4). This part is needed for the implementor of the pattern catalog only. The user of PSiGene does not need the Implementation part as the system does all the necessary implementation work.

Some of the patterns currently implemented in PSiGene (see table 1) are nearly identical to more abstract patterns presented in other catalogs, some are domain- and appli-
Pattern ThermalMass

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

Also Known As
Simulation thermischer Masse /1/

Motivation
A volume has to act as a thermal mass to compute its temperature. This pattern works...

Applicability
This pattern can be bound to any thermal mass. This is typically a room or a thick wall...

Structure

<table>
<thead>
<tr>
<th>target</th>
<th>compute ...</th>
<th>sinkSource</th>
<th>connection</th>
<th>heatFlow ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volume</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heat source</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heatFlow ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Participants
- **target**: Object
  Object to bind pattern to.
- **temperature**: Attribute
  Last computed temperature.
- **volume**: Attribute
  Volume of the thermal mass.
- **compute**: Method
  Does the calculation cycle once.

Collaborations
...

Consequences
...

Implementation
Smalltalk-Code-Templates

```smalltalk
{compute}
| heatCapacity collectedHeatFlows
| timeNow |
| timeNow := Scheduler simSched

| simMillisecondClockValue. |
| heatCapacity := self |
| (getHeatCapacity); |
| collectedHeatFlows := self |
| (collectHeatFlows). |

| self |
| (calculateTemperatureWithCapacity: |
| heatCapacity |
| withHeatFlows: |
| collectedHeatFlows |
| while: (timeNow - self |
| (timeOfLastComputation)). |
| self (timeOfLastComputation): |
| timeNow. |
```

Related Patterns
Thermal Exchange

cation specific adaptations of well-known patterns. Examples are SingleIndirection and ContinuousComputation which roughly correspond to Adapter and Mediator from /GHJ95/. In addition, we derived some highly domain dependent patterns from prototype building simulators, for example ThermalMass (figure 2) which describes the thermal calculation of a volume with respect to heat sources and connections to other volumes.

The patterns and the catalog form a pattern system /BMR96/: every pattern provides just a partial solution to the simulation problem. In order to form a working simulator, all patterns must work together (e.g. ThermalMass must cooperate with ThermalJunction, ThermalExchange, and ContinuousComputation). This problem is addressed in two steps: first of all, all patterns are able to work together with certain other patterns
by construction. Second, the formal interface definition of the patterns is used to ensure that all bindings are complete and correct.

**Table 1**: Some available patterns from the catalog

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Effects</td>
<td>ThermalMass</td>
<td>calculate walls or rooms thermally</td>
</tr>
<tr>
<td></td>
<td>ThermalJunction</td>
<td>calculate the heat exchange through different materials</td>
</tr>
<tr>
<td></td>
<td>ThermalExchange</td>
<td>calculate the transition from gas to solids and vice versa</td>
</tr>
<tr>
<td></td>
<td>Radiator</td>
<td>Calculate heating installation</td>
</tr>
<tr>
<td>Simulator Control</td>
<td>ContinuousComputation</td>
<td>simulate continuous time domain problems</td>
</tr>
<tr>
<td></td>
<td>EventTriggeredComputation</td>
<td>discrete event simulation</td>
</tr>
<tr>
<td></td>
<td>Sensor</td>
<td>monitor values</td>
</tr>
<tr>
<td></td>
<td>Actuator</td>
<td>modify values</td>
</tr>
<tr>
<td></td>
<td>UpdateControl</td>
<td>react on changes</td>
</tr>
<tr>
<td></td>
<td>ConcurrentProcess</td>
<td>compute concurrently</td>
</tr>
<tr>
<td>Structural Adaption</td>
<td>SingleIndirection</td>
<td>transport values from one object to another</td>
</tr>
<tr>
<td></td>
<td>DistributedSum</td>
<td>gather and add up different values</td>
</tr>
<tr>
<td></td>
<td>MethodAlias</td>
<td>map between names</td>
</tr>
</tbody>
</table>

The complete catalog is divided into three categories: patterns for the calculation of physical effects (e.g. ThermalMass), patterns to control the simulator (e.g. ContinuousComputation), and patterns to deal with different structures in the class model (e.g. SingleIndirection). The purpose of the catalog is to document the available patterns and their usage for the user of PSiGene. An implemented version of the catalog is used within PSiGene as described in the next section.

**4. Pattern-Based Generator**

All patterns provided by the catalog are realized within PSiGene as VisualWorks classes. The implementation ignores the informal parts of the pattern description (e.g. Intent) and emphasizes the Participants as well as the Implementation part. The implemented patterns are descendants of a common superclass AbstractPattern. This class provides the protocol needed to bind patterns, to generate code, and to create error reports for the user. As with ordinary classes, patterns may be either abstract or concrete. Abstract patterns provide just a partial protocol, for example the code templates may be missing. Patterns may be arranged in an inheritance hierarchy if specializations of general patterns are needed. Also, patterns may aggregate other patterns in order to use their services. Technically, this means a pattern may bind other patterns to the class model. For these reasons, the pattern class hierarchy within PSiGene is not identical with the patterns from the catalog: in particular, there are some smaller patterns (or idi-
oms), which do not appear in the catalog, but which are aggregated by other, more complex patterns within PSiGene.

PSiGene assumes that the class model for the simulator is already implemented (by the ADT generator, see figure 1). Nevertheless, this class model and eventually the simulation objects are inputs to the generator, too (figure 1). The generation of simulation code is then divided into two phases, which are performed automatically, without manual intervention:

In the binding phase, an instance of a pattern class is created for every pattern bound to the simulator's class model. In this phase, no code is generated. The set of all instances can then be checked for completeness, based on the formal parameters defined by each pattern. A report is generated for the user of PSiGene. This report includes information about missing and wrong bindings for every single pattern instance so that errors in the binding description can easily be identified and corrected.

In the generation phase, all pattern instances are asked to generate code for the objects they are bound to. There is no 'master generator': each pattern instance is responsible for its own code portion (by providing a code generation method), making the generation process flexible and easily extendible.

The main task of the generation phase is to select, personalize and instantiate the code templates of all pattern instances. PSiGene supports multiple code templates for every method to be implemented. The first step during code generation is therefore the selection of the right template, based on the pattern binding information and the class model. The code must then be personalized by replacing generic code fragments (macros) with the information derived from their actual binding. After that, the code has to be instantiated, for example by adding a method to the target class the pattern is bound to.

Code templates are defined as source code fragments with macros for the necessary adaptations (see figure 2). The macros are placed in braces. They may be used for simple elements like variable names, but they may also represent more complex code fragments like loops, which is important for optimizations.

A very simple pattern binding example is shown in figure 3. A room needs to know the amount of heat $q$ from all radiators. The pattern $\text{DistributedSum}$, based on its binding, is used to install a simple method in class $\text{Room}$ which performs the calculation. The simplified generation method for $\text{DistributedSum}$ looks as follows:

```plaintext
generateCode

| macroText |

... macroText := self perform: \#sumTemplate.
macroText replaceMacrosUsing: self macrosAndElements.
self putMethod: (self elementAtMacro: 'sum')
  withSource: macroText
to: (self elementAtMacro: 'target').
```

For more complex patterns, the generation method selects the kind of code linking (see below), performs optimizations, assembles methods from smaller fragments, compiles finite state machine definitions, and aggregates other patterns. If the class model does
not match to the pattern’s Structure, a special set of patterns is used to transport information to the desired objects.

The code may be linked to the simulation objects in three different ways:

- **Inheritance:**
  An additional superclass is added to the simulation object’s class. This is the easiest way to link the simulation code. But it has some drawbacks: the inherited code is generic and can not be optimized for the simulation object. Multiple inheritance may impose some problems and is not available in all languages.

- **Delegation:**
  An instance of a special pattern-implementing class is created. This also does not permit individual code optimizations. But it can avoid problems related with multiple inheritance. The ContinuousComputation pattern uses this technique to provide multi-threading capabilities for simulation objects.

- **Code injection:**
  The code is installed within the simulation object’s class. This allows powerful optimizations of the code for individual classes, but may lead to a large amount of code if a single pattern is instantiated many times. Nevertheless, this is the most common way to create code within PSiGene.

Which method exactly is used depends on the target language, the kind of pattern, and on the context of the pattern usage. The decision is made by the code generator.
A Brief Example
In this section, we will examine a very simple example for the thermal simulation of a single room with one radiator as heating installation and walls to the environment at all sides (no windows, doors, lighting installation, users, and so on). The simulator kernel is available as a library, presented to the user as a class model (see screen snapshot in figure 4). It provides all classes needed to provide the basic simulation services.

Fig. 4 Screen snapshot of the simulator kernel library

The first step one has to do in order to create a generator is the definition of the class model for the simulation objects. In this case, we primarily need a room, a representation of a wall and a surface between room and walls (modeled as a sequence of layers), see screen snapshot in figure 5. This figure shows the object types and the object relations. Lines with a dot at the end denote inheritance. Object attributes are not displayed graphically. Let's assume that attributes are assigned 'reasonably'. Lines leaving the diagrams to the right are relations to the simulator kernel from figure 4. This model is input to the ADT generator of MOOSE, which in turn creates the complete class hierarchy with all attribute and relation access functions.

In a second step, the user has to select the appropriate patterns from the catalog (which is not supported by PSiGene) and has to specify the pattern bindings. At the moment, this is done textually, but we are currently implementing a graphical binding editor which will allow the user to bind patterns directly to the class model. This editor will include the possibility to check the pattern bindings and to display wrong bindings on
the screen, which we hope will greatly simplify the process of setting up correct bindings. A part of the example's bindings can be seen in the following code fragment:

```
"--- RadiatorSurface ---"
ThermalJunction
bind: 'target' to: 'RadiatorSurface';
bind: 'area' to: 'area';
bind: 'thermalResistance' to: 'thermalResistance';
bind: 'getTemperature' to: 'temperature';
bind: 'getThermalResistance' to: 'getThermalResistance';
bind: 'getHeatFlowFor' to: 'getHeatFlowFor';
bind: 'neighbouringObject' to: 'surface'.

"--- Radiator ---"
ContinuousComputation
bind: 'target' to: 'Radiator';
bind: 'activeObject' to: 'activeObjectOfRadiator';
bind: 'event' to: 'eventOfRadiator';
bind: 'compute' to: 'computeRTemp';
bind: 'interval' to: '10000'; "calc every 10000ms"
bind: 'priority' to: 'self priority'.

....

"--- Surface ---"
...... in principle like RadiatorSurface
"--- HeavyLayer ---"

....
ContinuousComputation
bind: 'target' to: 'HeavyLayer';
bind: 'activeObject' to: 'activeObjectOfHeavyLayer';
bind: 'event' to: 'eventOfHeavyLayer';
bind: 'compute' to: 'computeThermalMass';
bind: 'interval' to: '30000';
bind: 'priority' to: 'self priority'.

Fig. 5  Screen snapshot of the example’s simulation object class model
Let's take a closer look at one of the bindings: for class Room, the pattern ContinuousComputation is bound to several methods, attributes and relations. The target for the pattern is the class Room itself. The activeObject (from the kernel library, implementing multiple threads) can be found following the activeObjectOfRoom-Relation, event is handled in a similar manner. The method that needs to be computed is named computeThermalMass, the reactivation interval is 30000ms. Process priority is standard priority.

As one can see from the binding code fragment, the patterns serve as glue between different simulation objects, defining their interaction (e.g. the ThermalMass pattern bound to Room and, via DistributedSum, also bound to Surface and RadiatorSurface), and they serve as glue between the simulation objects and the simulator kernel classes (e.g. the ContinuousComputation pattern bound to Room and to kernel's ActiveObject via activeObjectOfRoom).

In the next step, simulation code is generated. PSiGene reads the class model for the simulator kernel and the example classes (see figures 4 and 5) and the pattern bindings.

For every pattern instantiated in the bindings, it creates an instance of the pattern class implemented within PSiGene. When all patterns are instantiated, the resulting set of instances is checked for completeness and correctness. Afterwards, the generation...
method is called for every pattern instance, which in turn creates the application code. For the pattern ContinuousComputation bound to class Room, this means that two methods are generated, an init method called by the constructor, and an establish method, which activates the periodic computation and looks as follows:

```plaintext
establish
|aThread|
... "some initializations"
"create Thread;"
aThread := Thread createThread:
[ :currentEvent |
  self computeThermalMass.
  (self eventOfRoom instance)
  timestamp: currentEvent timestamp + 30000;
  deadline: currentEvent deadline + 30000;
  send.
] atPriority: self priority.
"register Thread within ActiveObject;"
self activeObjectOfRoom instance addThread: aThread
  reactOn: #activate.
self eventOfRoom instance send.
```

All parts of the method where template macros were replaced with the actual bindings by the generator are underlined. For this simple example, code generation means simply macro replacement. However, PSiGene is also able to create much more complex code (for example, to compile finite state machine definitions into methods), or to perform more elaborate macro replacements, for example to eliminate loops over sets if it is known in advance that the cardinality of the set is exactly one.

5. Related Work

This work is closely related to works in the area of application generators and pattern-based design strategies.

Our approach shares with application generators (e.g. UI builders, simulation systems) that it is always optimized for one domain. However, we rely on object-oriented modeling and the pattern approach and we believe that our method in itself is usable in many domains, only concrete manifestations of the method are domain dependent. In contrast to many application generators (for example from the simulation domain), we don’t use domain specific modeling languages (such as discrete event simulation languages) because we wanted the user of our method to be an expert in object-oriented design, not in simulation techniques.

Code generation in PSiGene is a one-step approach. There are other approaches to code generation, based on program transformation (see for example GenVoca /BST94/ or CIP /GoH85/). There, an abstract program is transformed to implemented code by a sequence of refinement steps. The input for these types of generators is usually more abstract than our models. However, the goals of this approach (in terms of productivity gains, reuse potential, scalability, and optimizations) are the same. Although GenVoca has a different focus (component based), we share its basic principles (see /BST94/): generation from subsystem building blocks, standardized interfaces, and parameterization.
At the moment, many people experiment with pattern-based design methods and with design patterns. We share with these approaches the idea of having a design catalog, capturing good design in general /GHJ95/ or domain specific knowledge (see examples in /CoS95/). Also, some people discuss pattern languages or pattern systems in order to make patterns work together. A discussion of pattern systems, as well as a complete system, can be found in /BMR96/. As far as we know, not much work has been done in automatically generating code from pattern descriptions. In /BFV96/, an approach is described which creates some code from a partially automated, WWW-based pattern catalog. However, this approach does not reach the amount of integration of object-oriented structure models and connected patterns as our approach does. Finally, there is at least one approach which uses design patterns for the same purpose as we do: in the DEMETER project /Lie96/, objects and operations are kept separate and design patterns are used to adapt to different object-oriented structure models. Operations can be performed on objects even if the model changes. This approach does not use code generators, the mapping is performed at runtime.

6. Discussion of our Approach

Our application domain, building simulation, has two characteristics we exploited for the development of our method. First of all, we wanted to be able to generate large families of building simulators with different requirements, as stated in the introduction. Therefore, a generator based approach was feasible. Second, in this domain, we find a limited number of primitive operations and interacting objects, like solving differential equations, executing methods periodically, controlling objects with finite state machines, doing discrete event simulation, and so on; these operations stay always the same, they are just combined in different ways. Exactly these recurring design problems of combining the right operations in the right way were put into the patterns.

Setting up the pattern catalog is a difficult task: on one hand we would like to have patterns as abstract and powerful as possible. On the other hand, every pattern should be as flexible as possible regarding its usability for different class structures. We always have to find the right tradeoff between these points, as a powerful pattern, involving many classes and relations, depends very much on a certain class structure. This limits the possibility of using the pattern in a wide variety of situations and with different class models. There is no general rule of how to set up the pattern catalog. At the moment, we try to deal with this problem by allowing to define complex, specialized patterns hierarchically from smaller, flexible patterns.

The patterns were gained by a sequence of hand coded simulator prototypes which we developed. Therefore, the code generated by PSiGene is comparable to handwritten code. This is valid for the code length as well as for the performance. While experienced programmers might be able to write better code because of very ‘local’ optimizations, PSiGene does it’s job as good as average programmers. The overall code quality is still better, because the generator produces no programming errors, given that the patterns are correct. If a code template should have errors, fixing it can be done in minutes.

While it took nearly the same time to implement PSiGene as it took to implement one of the simulator prototypes, productivity gains dramatically using PSiGene. We con-
ducted an experiment where we produced a thermal building simulator by hand and with PSiGene. Class model (about 15 classes) and functionality, as well as the used physical abstractions were the same. The designers had comparable skills and knowledge. It took three weeks to implement the simulator by hand, using the ADT generator (figure 1). Using PSiGene, setting up the bindings (about 20 pattern instantiations from about 10 different patterns) took one day. Both simulators came up with about 5000 lines of code. Of course, this comparison only holds as long as we always find the required patterns. Installing a new pattern is done in minutes, however, developing a new pattern may take significant time, depending on its purpose.

Changing simulation requirements, e.g. changing the physical abstraction for walls, is also a matter of minutes, as long as the required patterns are present. For the hand coded version, it takes significant time to do the same.

There are some limitations of our approach: first of all, the user is restricted to a limited pattern catalog. The approach tends to produce large pattern catalogs with small patterns (in order to use the patterns for different class structures, see above), making the modeling less abstract. Here, it takes skilled pattern writers to find the right level of abstraction. Our approach develops its abilities best when the required code has a certain generic potential. If nearly every method is unique, we can not profit from instantiating the same pattern many times. Although modeling takes place on a rather abstract level, specifying the pattern bindings is still a complex and creative task: building simulation does not come for free. Last not least, we have (up to now) no possibility to debug pattern bindings. Finding errors in the bindings other than syntactical ones may be a hard task. It would be desirable to debug patterns graphically on the binding level, but further research is needed to find a satisfying solution.

Nevertheless, we believe that our approach is useful in many domains, particularly those where the characteristics are the same as mentioned above. If we take a look at table 1, we will find that primarily patterns of the 'Physical Effects' category are restricted to building simulation. Other Patterns like ConcurrentProcess or Actuator could very well be used for different problem domains like control systems. Here, the transition to the new domain would probably be straight forward. We are currently extending PSiGene to handle distributed systems and building control systems. Other technical application domains might match our approach as well.

7. Conclusions

We presented the PSiGene approach to generate application code for one domain from object-oriented models and patterns. Because object types and operations are kept separate and are connected via patterns, the code generator is very flexible with respect to changing class models and different patterns. Experiments showed that about 10 patterns (with about 20 instantiations) are sufficient to specify a framework for thermal building simulation. Every additional physical effect takes another 5 to 10 patterns. This shows that even for complex simulation problems the number of patterns and instances is quite moderate.

Because the major part of reuse takes place while adapting existing models and combining them with patterns, the potential of our method develops best when used in environments where the underlying models and/or requirements are rapidly changing.
and where the characteristics of our application domain (see section 6) are met, like many technical domains. This is particularly true for environments where applications have to exist in customized versions for every project.

Our pattern catalog is much more concrete than other catalogs and can not be used for all design problems in other domains. However, some of our patterns are just domain specific adaptations of more general patterns, and we believe that domain specific pattern catalogs are an effective way to support the designer.

All in all, our approach leads to an efficient way of generating code for certain types of applications. All code generators within MOOSE (and therefore also PSiGene) are aware of manually written code: MOOSE/PSiGene can be combined with traditional OOD methods and manual extensions can be made wherever required. We believe that our tool is an effective help to application designers and programmers.

References


/GHJ93/ E. Gamma, R. Helm, R. Johnson, J. Vlissides: “Design Patterns: Abstractions and Reuse of Object-Oriented Design“, Proc. ECOOP ’93, 1993

/GHJ95/ E. Gamma; R. Helm; R. Johnson; J. Vlissides: “Design Patterns“, Addison-Wesley, 1995


/Pre95/ W. Pree: “Design Patterns for Object-Oriented Software Development“, ACM Press, Addison-Wesley, 1995
