Modeling Software Architecture Using Domain-Specific Patterns

J. P. Riegel, C. Kaesling, and M. Schütze
Dep. of Computer Science, University of Kaiserslautern, Germany,
{riegel, kaesling, schuetze}@informatik.uni-kl.de

Keywords: architectural design patterns, domain-specific modeling support, code generation

Abstract: In this paper we present a domain-specific modeling approach for application components. We use class diagrams and design patterns as major modeling notations and utilize code generation techniques to create an application. Certain architectural aspects of these applications can explicitly be modeled using concrete versions of architectural patterns. As an example, an adaptation of the Pipes and Filters pattern (see Buschmann et al., 1996) is presented, which can be used as an architectural modeling entity and which is supported by a code generator for automatic implementation of different data flow mechanisms.

1. INTRODUCTION

Software components are an important factor in software development. To successfully use a component, its architecture should match to the overall application architecture. This implies that the component architecture must be adaptable with respect to the needs of a specific application. The need for flexibility leads to the questions: "How can the architecture of a component be represented and influenced? Which parts of the software architecture are fixed, which can individually be modeled or varied? Is code generation for architectural aspects possible?"

We try to give an answer to these questions by capturing architectural elements with variants of design patterns and by providing modeling and code generation techniques that allow the user to influence and adapt a components architecture to specific needs. The work presented in this paper

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

is integrated into an experimental, domain-specific development method called PSiGene (*p*attern-based *si*mulator *gene*rator). The goal of PSiGene is to provide a powerful modeling environment to support the creation and integration of customized components. Our initial application domain is building simulation, but support for other domains is possible as well.

In our case, simulators are used in the domain of building automation to test control algorithms. Building simulators must exist in many variants to cope with various physical effects, combinations of effects, required accuracies, and different time advancement schemes (e.g., real-time, time-warp). One complex simulator can not fulfill all possible requirements at the same time, therefore tailored simulation components are required. PSiGene provides a pattern based modeling and code generation environment to support the development of customized building simulators. Section 2 gives a short introduction to PSiGene. For further readings see Schütze et al. (1997) and Heister et al. (1997).

In this paper we present an extension to our initial approach. In order to become more domain independent and to be able to handle more complex models, we emphasized the separation of different component aspects; i.e., we distinguished between component architecture and component behavior. The following figure (*Figure 1*) illustrates the engineering process of PSiGene.

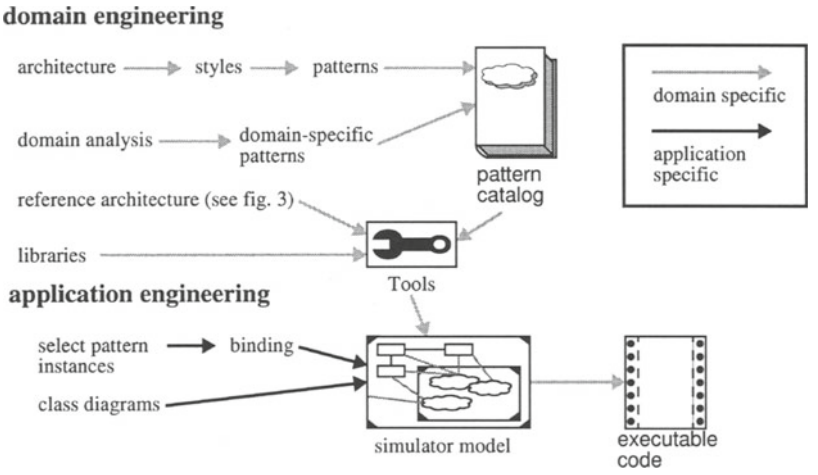

Figure 1. Domain- and application-specific tasks

Some parts of PSiGene, in particular the pattern catalog, the reference architecture, and the libraries are results of a domain engineering step. We tried to capture architectural styles for some component aspects in patterns.

They are designed to work together with domain-specific (behavioral) patterns. All patterns from the catalog form a system of patterns (Buschmann et al. 1996). In addition to the catalog, a reference architecture (see *Figure 3*) was set up, and supporting libraries have been implemented. To design a simulation component (application engineering), appropriate patterns have to be selected from the catalog, instantiated, and bound to class diagrams. Executable code is automatically generated for this application model and can be extended with manually written code if needed.

The following section gives a brief introduction to PSiGene. An analysis of this approach considering software architecture is found in chapter 3. After that, chapter 4 describes two of our architectural patterns (*Pipe* and *Filter*) and gives a short example of their use. A discussion of the approach and an outlook on future work conclude this paper.

2. PSIGENE

PSiGene is a component-based, domain-specific software development approach (for details see Schütze et al. 1997). It's purpose is the creation of tailored, application specific components: in contrast to many component based development methods, where components are provided "as is", PSiGene represents a flexible meta component. The user of PSiGene specifies the concrete component with a model, a generator implements the component automatically from this specification. This results in the creation of components that exactly match the applications needs without introducing any overhead in runtime or memory consumption caused by generic code or interpretation of runtime parameters.

PSiGene combines object-oriented modeling of the static aspects of a component (class diagrams) with pattern-based modeling of the dynamic aspects like component behavior or functionality (pattern instance models), and with code generation techniques for the implementation. The initial application domain of PSiGene is real-time simulation of large buildings.

PSiGene does not work stand-alone, but is integrated into a larger software development environment called MOOSE (*m*odel-based, *o*bject-*o*riented *s*oftware generation *e*nvironment). Within MOOSE, every application consists of a set of components each implementing one aspect of the overall application features. An application is defined by an application model, which in turn consists of several component models. A set of domain-specific generators is used to transform the models into software components. A certain type of generator, the so-called cross-component generator, is capable of interpreting more than one component model at a

time and of generating glue logic and application interface code from the interrelations (which we call the "glue") between different component models. PSiGene's generator is implemented as a cross-component generator within MOOSE. More details about MOOSE can be found in Altmeyer et al. (1997).

2.1 System Overview

Figure 2 gives an overview of the implementation of PsiGene. An application, in this case a building simulator, is defined with an application model. Among the different component models we find a structure model (expressed as a class diagram by using editors from MOOSE) that defines the simulation objects. Other class models define structures for other aspects of the simulation or represent run time libraries.

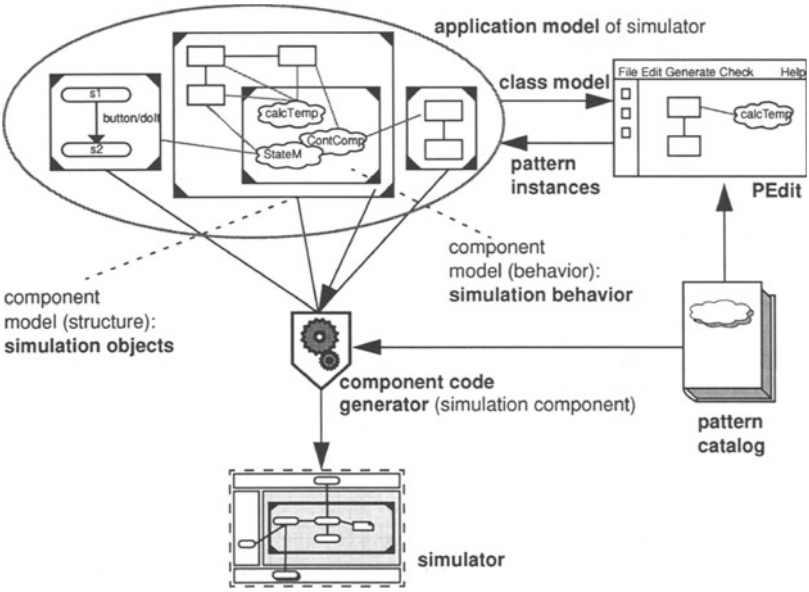

Figure 2. PSiGene overview

The behavior of the simulator is defined with a pattern instance model. Patterns, which are taken from a catalog (see below), are used to define the behavior of the simulation objects, to define the overall functionality of the component, or to define the interface between the simulation component and other components expressed by other models. These pattern instances not

only specify local component properties, but also the glue to other components. This means that component integration is also performed on the modeling level. The pattern instances are created using a graphical editor, PEdit, that displays class models and lets the user select and instantiate patterns from a catalog. These instances are then bound to the class model, which means that each instance is connected to elements (classes, relations, attributes, methods) of the class model.

Once the application model is set up, it is fed into PSiGene's generator. The generator reads the structure model, the pattern instances, and it knows the patterns of the catalog. From this information, it creates optimized, tailored component code. For variants of the application, we will simply generate a variant of the component code. Details of pattern-based code generation can be found in Heister et al. (1997).

2.2 Pattern Formalization and Pattern Catalog

As explained before, patterns used for modeling are taken from a domain-specific pattern catalog. The intention of the catalog is pretty much the same as with other pattern-based design methods: to capture successful, "good" design and to provide this knowledge to the catalog user by presenting solutions for smaller design problems in a certain design context. One of the first and most famous catalogs has been presented by the "gang of four"; see Gamma et al. (1995). In contrast to this and most other catalogs found in the literature, which address general design problems, we focus on concrete design problems for building simulation such as the calculation of heat flows in buildings or the scheduling of real-time processes.

Table 1 shows the structure of our catalog. It is partitioned into several categories dealing with different (orthogonal) aspects. As an example, some patterns from each category are shown.

Because we set up the catalog for a very narrow application domain, we are able to state the problems as well as the solution very precisely, enabling tool support for modeling as well as code generation. At the same time, we had to formalize the pattern approach with respect to the pattern interface and the code templates provided as problem solution: In contrast to other approaches, we have to specify the binding between the class model (structure model) of the application and the pattern instances formally and unambiguously. And we need code templates that are suitable for code generation.

Within PSiGene's catalog, the pattern interface, defining the structure as well as the participating elements of a pattern, is expressed with *name:type* pairs as formal parameters. The *name* denotes the name of the participating element, the *type* shows which parts of other component models are eligible

for binding, e.g., classes, relations, methods, and so on. Based on the formal parameters, the (syntactical) correctness and completeness of pattern bindings can be checked by tools. With that, the formalization builds the syntactical framework for a pattern language, as the cooperation of patterns can be expressed with formal bindings. Furthermore, the code generator gets sufficient information to create component code.

Table 1. Excerpt from the pattern catalog

Category	Sub-Category	Patterns	Description
Framework Structural Adaptation	Primitive	VariableValue	Access an attribute
		BufferedValue	Attribute buffer that is mainly used in conjunction with a Pipe
	Indirection	FollowRelation	Delegation along a relation
		Traversal	Collect connected objects without specifying a path
	Redirection	MethodBranch	Branch if condition is met
	Pipes and Filters	Pipe	Specify data flow
		Filter	Activity when using a Pipe
	Distribution	AttributeProxy	Used for distributed access
Simulation Control	Control	Actuator	Set attributes with events
		ContinuousComputation	Periodic method invocation
	State Machines	StateMachine	Simple finite state machine
		StateMachineActive	State machine using conditions
User Interface	Display	DisplayAttribute DisplaySlider	Display an object's attribute Display attribute as a slider
Domain	Simulation	ThermalMass	Calculate temperatures
		ThermalJunction	Compute heat flows

The code templates are split into smaller fragments. Each fragment consists of code in a given programming language, enriched with macros that denote the variable parts of the code. Currently, we support Smalltalk as the target language, however, provisions have been made to generate code for other object-oriented languages as well. During code generation, the generator collects the fragments, "personalizes" them by replacing the macros, and assembles the resulting code to methods. Macro replacement can be as simple as string exchange or it can mean to replace a macro with other, complex code fragments recursively. The definition of replacement

strategies and code fragments is supported by inheritance and by pattern aggregation.

2.2.1 Extensions to PSiGene

Components generated by PSiGene do not work in isolation, but are embedded into a surrounding application with an underlying software architecture determined by the application domain and other forces. For earlier versions of PSiGene, this application architecture was fixed, and consequently, the component architecture was fixed, too. There were architectural aspects that have been addressed by PSiGene, e.g., the degree of multithreading in a simulator or the possibility to create distributed simulators. However, the decision about architectural elements has been made implicitly, while choosing patterns that determined other simulation aspects. For example, by using the Sensor and Actuator pattern to simulate hardware interfaces, the user implicitly enabled distributed simulation and influenced the component's and application's interface. As we started to apply PSiGene to other application domains, we realized that our approach would become more general and the modeling would be significantly easier if we were able to specify the architecture of applications explicitly. The following section will illustrate how we adapted the latest version of PSiGene (in particular the pattern catalog) to capture and model architectural styles, and how we generate code that implements these styles automatically from the models.

3. SOFTWARE ARCHITECTURE WITHIN PSIGENE

The architecture of a software system can be modeled following architectural styles (see Buschmann et al., 1996, and Bass et al., 1998). Styles give concrete hints on how to construct and organize a system. For example, following the Client-Server style leads to a system where several clients communicate with one or more servers. The exact behavior of a specific client or server is independent of the architectural style and must be specified separately. Tracing which style leads to which component structure makes the software more maintainable and understandable.

Usually several styles can be identified in a component's architecture. Each style can be seen as a set of constraints on an architecture. These constraints define a family of architectures that satisfy them (Bass et al., 1998, p. 25). Some of these constraints can also be expressed with design patterns (compare Monroe et al., 1997). Such a pattern includes the context

in which a style can be applied, the forces it resolves, the consequences, and the structure of the style. In addition to this, patterns contain a guide on how to apply them.

Finding concrete patterns that reflect an architectural style is not easy: styles are an abstract description of facets of a software architecture, whereas (PSiGene-) patterns are usually applied to smaller parts of a component and reflect concrete design decisions rather than organizational structures.

We formalized some architectural patterns so that they can be used within PSiGene. Their binding enforces a certain architecture and code can automatically be generated. The main drawback of this "formal" description of architectural styles is that PSiGene patterns cannot capture the whole bandwidth of possibilities how a style can be implemented: only a limited number of domain-specific implementation strategies can be included in a single pattern because otherwise code generation would be impossible and the binding would become far to complex. This restriction, however, doesn't count as much, because our patterns don't aim to be universally applicable but are only used in one domain. When focusing on one domain, architectural styles occur only in few variants.

As explained before, the previous version of PSiGene used architectural styles mostly implicitly: the patterns concentrated on solving a certain (simulation) problem and therefore they contained behavioral aspects as well as structure and other architectural components. For small models this was convenient, but when modeling complex simulators or when adapting PSiGene to other domains it is desirable to be able to model the architecture more explicitly. To do so, we reengineered some of our patterns and added new ones to reflect certain properties of architectural styles.

3.1 Architecture in PSiGene

All simulation components that are modeled with PSiGene share a common basic architecture. Some parts of this architecture are fixed while other parts can vary from simulator to simulator. *Figure 3* gives an overview.

A set of fixed components builds the framework that houses customized simulation components. The framework is used by inheriting from or delegating requests to framework objects or classes. Three major components are used: a GUI library to display simulated objects and to stimulate the simulator, an I/O library to communicate with other applications and to log simulator runs, and the kernel library which is responsible for scheduling and event-handling. The structure and behavior

of the simulation components, however, varies for different simulators in order to match the needs of the applications. Variable aspects are:

- component structure (i.e., class models)
- component functionality and behavior
- non-functional requirements (e.g., timeliness, accuracy)
- component integration: glue code to connect to the framework

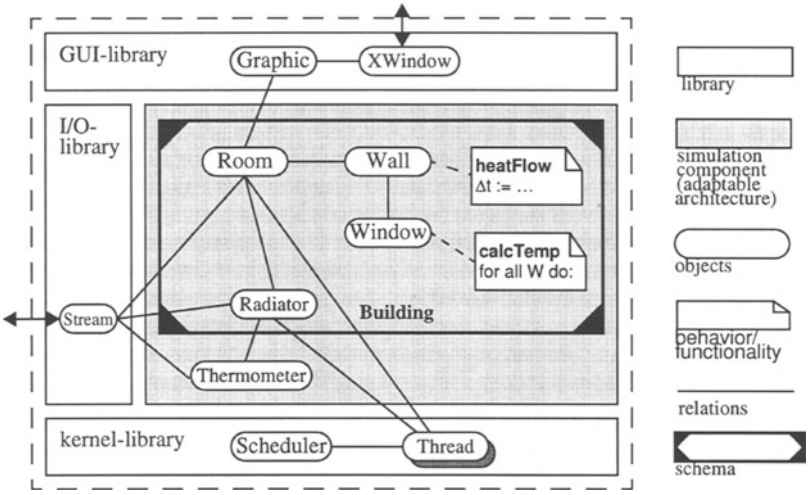

Figure 3. Architecture of a building simulator

3.1.1 Architectural styles in PSiGene

Several architectural styles are used to model a building simulator. The following table (*Table 2*) gives an overview of the styles that occur in PSiGene.

Two styles, Repository and Pipes and Filters describe data aspects of the model. Our components are modeled using class diagrams. Different components can share parts of these diagrams to have access to the same data. Methods to access such a data repository are automatically generated. Data exchange within one component is modeled with the *Pipe* and *Filter* patterns. The communication channels are seen as pipes, and activities to trigger the data flow are modeled as filters (see next chapter).

The application framework implements the framework style. Framework components are represented by class diagrams and can be incorporated into the models using object-orientated mechanisms and patterns.

Since our kernel library is event-driven, all active simulation objects must be able to receive and evaluate events. Event handling is also modeled with patterns (such as *Actuator* or *ContinuousComputation*).

In this section we have shown how the architecture of a simulation component looks like and which architectural styles occur implicitly by using PSiGene. Some styles can also be expressed explicitly with patterns, as we will illustrate with the example in the next section.

Table 2. Architectural styles in PSiGene

Architectural Style	Occurrence	Modeling Notation / Support
Repository	Data exchange between components	Class diagrams
Pipes and Filters	Specify data flow between simulation objects and identify active objects	Pipes and Filters patterns
Framework	Application framework	Class diagrams, schemas, patterns
Layers	Accessing libraries (via delegation)	Indirection, control, and display patterns
Model-View-Controller	Used in the GUI library. The 'Model' is part of the simulation component	GUI patterns
Distribution / Event Systems	Network communication with other applications (I/O library) or distributed simulation/scheduling (kernel library)	Patterns and library parameters
Microkernel	Useful to encapsulate communication aspects esp. in the kernel library	Patterns plus hierarchical class diagrams (not yet supported by PSiGene/MOOSE)

4. EXAMPLE

Up to now, the software architecture of our building simulator models was defined by the framework: the libraries, the structure of our patterns, and by the way the class diagrams are constructed. Many of the patterns addressed behavioral aspects as well as other software architectural aspects.

For example, the *ThermalJunction* Pattern is used to simulate the junction of two adjoining thermal masses. A thermal mass is a simulation object that has a relevant heat capacity. Examples are rooms, radiators, or the environment. A thermal junction is typically a wall or a window. When two thermal masses are adjacent, they exchange energy through heat flows. The *ThermalJunction* pattern can be used to calculate the heat flow between any two of those masses. The heat flow depends on the difference of temperatures of the adjoining thermal masses and on the thermal resistance

of the separating (i.e., insulating) material. The first version of the *ThermalJunction* pattern assumes that it can somehow access the required temperatures and the thermal resistance by calling a method. Other patterns like *FollowRelation* or *Traversal* must provide this access methods (usually by delegation to appropriate objects).

Figure 4 shows a part of the model for the simulation of heat flows through a simple wall. The class diagram describes how rooms are connected via surfaces and walls. The lower part of *Figure 4* shows the pattern instances. *ThermalJunction* is bound to the class *Surface* and implements the calculation method. To collect the data for this calculation, several *FollowRelation* patterns are required. The temperature of both neighboring rooms has to be collected and the cumulative thermal resistance of the wall and both surfaces must be computed.

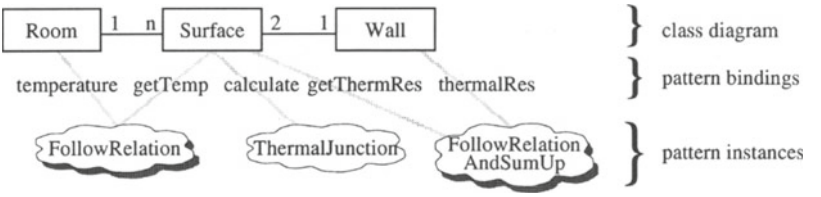

Figure 4. Simulating heat flow

ThermalJunction implements an action (calculation of the heat flow) that is closely related to a data flow (collecting temperatures and thermal resistances). *ThermalJunction* concentrates on the action part and also assumes the required data are present in a certain way. For small object models this is adequate as data flow is relatively simple. As models grow more complex, software architecture becomes more and more important. For the "thermal junction" problem this means, that the data flow aspect becomes more important (and more difficult to model) and the coupling between the data flow and the activity view has to be well considered.

Data exchange between simulation objects usually consists of two parts: a communication channel (object relations or possibly a network connection) and an activity that triggers the exchange. Such pipelines occur in many variants: push-driven, pull-driven, synchronized push/pull, distributed, buffered, and so on. To be able to model such a variety of different data flow possibilities, it is useful to decouple the data flow aspect from the functional aspects and model it separately.

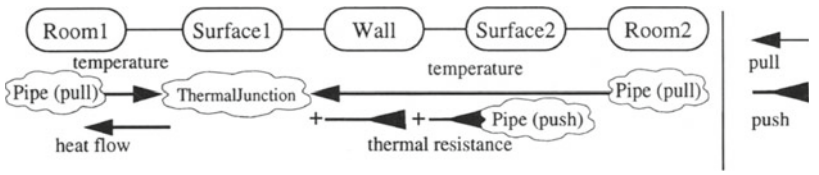

Figure 5. Data flow between two rooms

The new version of *ThermalJunction* focuses on the functional view only. The data to calculate a heat flow must still be present but the pattern doesn't prescribe how to access this data. Two new patterns, *Pipe* and *Filter*, can be used to model data flow. In our example (*Figure 5* and *Figure 6*), we have a data flow (i.e., a pipe) from the class *Room* to *Wall*, and an activity (i.e., a filter) to calculate the heat flow.

Whether the data flow is pull- or push-driven and/or distributed over more processes or computers is characterized by configuring parameters of the *Pipe* pattern. *ThermalJunction* can be seen as a *Filter* (from the data flow view) and bound to our *Filter* pattern (see *Figure 6*).

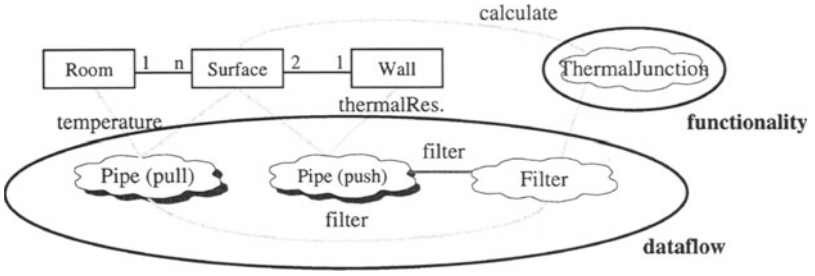

Figure 6. Different views for functionality and data flow

4.1.1 A Pipes and Filters pattern

This section describes our *Pipe* and *Filter* patterns in more detail. It is intended as an example of how software architecture can be expressed with PSiGene-like patterns. We took the pattern Pipes and Filters from Buschmann et al. (1996), which describes most properties of data flows as they occur in our domain (transfer, buffering, synchronization) and adapted it to our needs. The general static structure of a pipeline is shown in the class diagram of *Figure 7*. A pipe is used to connect a provider with one or more consumers. Push or pull methods are used to access data elements in the pipeline. Additional processing is done using filters.

Capturing the idea of the Pipes and Filters style in generative patterns is possible because the underlying structure is not too complex. However, dealing with the many variants in which pipelines occur is not trivial. We have realized the Pipes and Filters style as two individual patterns "*Pipe*" and "*Filter.*" They both implement a part of the Pipes and Filters structure (see *Figure 7*).

To identify pipelines in a class diagram, the patterns structure must be mapped to elements from that diagram. This structure mapping is done by assigning values to formal parameters of the *Pipe* and *Filter* patterns (see section 2.2).

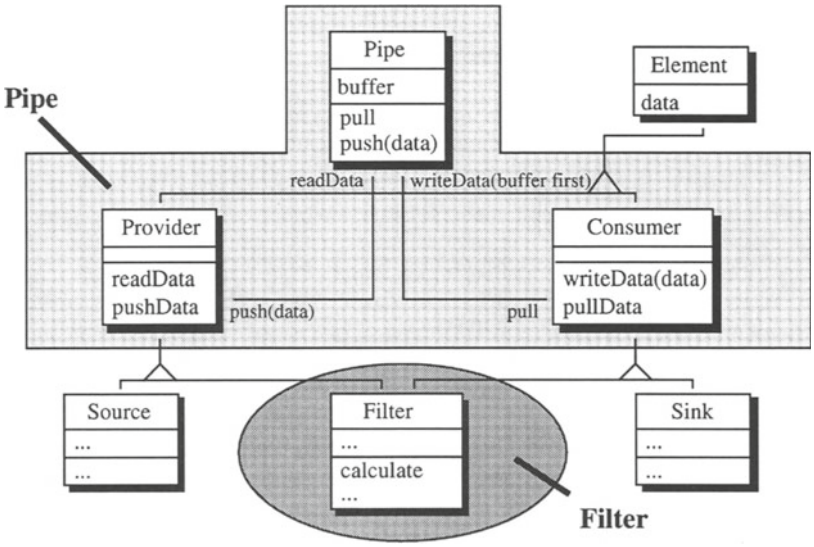

Figure 7. Object structure of the Pipes and Filters style (and pattern)

The following list shows the formal parameters of the *Pipe* pattern:

- **objects**:
  source
    the data source object
    (read as formal parameter *source:object*)
  destination
    the data destination object (sink)
- **attributes**:
  sourceData (use at source)
    the attribute that serves as the source for the data transfer

destinationData (implement at destination)
the new proxy attribute that is the sink of the transfer

- **relations**: entry and exit relations for the pipeline
- **methods**:
  read (optional, implement at destination)
  reads data from the source and writes it to the sink.
  push (optional, implement at source)
  triggers a data transfer. The initiator is the source object.
  pull (optional, implement at source)
  triggers a data transfer. The initiator is the destination object.
  notify (optional, use at destination)
  this method will be invoked at the destination object, if the data at the source has changed.
  request (optional, use at source)
  this method will be called at the data source if the destination object requires an actual value of the attribute. The source object has to transfer the current value (if it has changed since the last time).
- **properties**:
  bufferSize (optional, preset)
  if this property is set, a buffer is realized with the specified size.
  useProxy (optional, preset)
  this property instructs the generators to allow distribution of the participating objects over host-boundaries. A proxy mechanism is implemented.

As one can see, some parameters are optional and don't have to be bound. For example, a push-driven pipe does not need to bind the *"pull"* parameter. The *Filter* pattern is described by similar means. It is bound to calculation methods with a formal parameter *calculate:method(use)*.

Data flow aspects are modeled independently of other aspects. Interaction occurs only at well defined points. Functional patterns like *ThermalJunction* or *ThermalMass* are bound at the filter component using the formal parameter *"calculate."* Activity patterns like *ContinuousComputation* can be bound using the parameters *"pull"* or *"push"* from the pattern *Pipe*, or using the optional filter parameter *"compute."*

A small example demonstrates the pattern bindings for the model in *Figure 6*. After binding values to the formal parameters for all pattern instances, a binding description file is created. It looks as follows:

```
"ThermalJunction 1 - calculates the heat flow through a thermal junction element"
ThermalJunction
   bind: 'target' to: 'Surface';
```

```
    bind: 'calculate' to: 'calculateHeatFlowForRoom';
    bind: 'thermalResistance' to: 'thermalResistance';
    bind: 'area' to: 'area'.

"Filter 1 - calculates the heat flow through a thermal junction element integrating the pattern
    instances ThermalJunction1 and Pipe1 to Pipe5"
Filter
    bind: 'target' to: 'Surface';
    bind: 'calculate' to: 'calculateHeatFlowForRoom';
    bind: 'request' to: 'requestHeatFlowForRoom';
    bind: 'arguments' to: #('temperatureOfRoom' 'temperatureOfSurface' 'resistanceOfRoom'
    'resistanceOfSurface' );
    bind: 'getArguments' to: 'argumentsHeatFlowForRoom';
    bind: 'notify' to: 'notifyAttributeChangedForHeatFlowForRoom';
    bind: 'result' to: 'heatFlowForRoom';
    bind: 'initValue' to: '0.0'.

"Pipe 1 - provides the thermal resistance of a room at a connected surface"
Pipe
    bind: 'source' to: 'Room';
    bind: 'destination' to: 'Surface';
    bind: 'sourceData' to: 'thermalResistance';
    bind: 'read' to: 'readResistanceOfRoom';
    bind: 'exit' to: 'radiatorSurfacesOfRoom';
    bind: 'entry' to: 'roomOfradiatorSurface';
    bind: 'destinationData' to: 'resistanceOfRoom';
    bind: 'push' to: 'pushThermalResistanceToSurfaces';
    bind: 'notify' to: 'notifyAttributeChangedForHeatFlowForRoom'.

"Pipe 5 - provides the calculated heat flow from Surface to Room"
Pipe
    bind: 'destination' to: 'Room';
    bind: 'source' to: 'Surface';
    bind: 'exit' to: 'roomOfSurface';
    bind: 'entry' to: 'surfacesOfRoom';
    bind: 'read' to: 'readHeatFlowFromSurface';
    bind: 'destinationData' to: 'heatFlowFromSurface';
    bind: 'sourceData' to: 'heatFlowForRoom'.
    bind: 'pull' to: 'pullHeatFlowFromSurface';
    bind: 'request' to: 'requestHeatFlowForRoom'.
...
```

There are 3 objects classes in this example: a *Room* is connected with a *Surface* to a *Wall*. The instances of *Room* have to recalculate their temperatures in fixed time intervals. The rooms request the calculation of the heat flows from the adjoining objects indirectly by reading the local attribute "heatFlowFromSurface" (the calculating filter for the temperature at *Room* calls pullHeatFlowForRoom first, before accessing the attribute).

Since the generators know that rooms and surfaces are connected by a one-to-many relation, this attribute (implemented by "Pipe 5") holds a collection of heat flow values. Each of these values is calculated by an instance of the *ThermalJunction* pattern, the calculation is controlled by "Filter 1". This filter collects all required arguments, triggers the calculation while providing those arguments, and, if necessary, delivers the result via "Pipe 5". The following code fragment was generated from the above bindings:

```
argumentsHeatFlowForRoom
    "Collects all arguments for the calculation of heatFlowForRoom"
    | args |
    args := Array new: 4.
    args at: 1 put: self temperatureFromRoom.
    args at: 2 put: self temperatureFromWall.
    args at: 3 put: self thermalResistanceFromRoom.
    args at: 4 put: self thermalResistanceFromWall.
    ^args

computeHeatFlowForRoom
    "Does the calculation and stores the result in heatFlowForRoom"
    ^self heatFlowForRoom: (self calculateHeatFlowForRoom: self
    argumentsHeatFlowForRoom)
```

In our example the access to "heatFlowFromSurface" is triggered by a separate pull method (Pipe 5 is pull-driven). The other pipes are push-driven, which means that the data transfer is initiated by the sources of the pipe.

As one can see, our patterns "*Pipe*" and "*Filter*" realize a flexible data flow mechanism with synchronization capabilities. They separate this aspect from the functionality, which in this case is handled by *ThermalJunction*. *ThermalJunction* in turn does not care about data flow issues.

Depending on the binding, different transport and synchronization mechanisms can be implemented by *Pipe* and *Filter*. Some synchronization combinations are shown in *Table 3*.

Table 3. Some possible combinations of pipes and filters

Argument Pipes	Filter	Result Pipe	Comment
push-driven	inactive	push-driven	Each time a new argument is delivered, the result is calculated and propagated.
pull-driven	active	push-driven	The calculation of the filter is triggered by an external activity. The arguments are requested and the result is propagated.
pull-driven	inactive	pull-driven	If someone requests the result, it is

Argument Pipes	Filter	Result Pipe	Comment
			calculated after requesting all required arguments.
push-driven	active	pull-driven	This is the synchronized combination of the first three examples

The required synchronization mechanism depends on the frequency of data changes and on how the transport is triggered. The pattern interface allows to abstract from the concrete transport mechanism. Distributing the pattern or buffering values can be achieved by binding additional patterns like *AttributeProxy* or *BufferedVariable*.

4.1.2 Code generation

Each pattern instance in PSiGene comes with a partial code generator. It is responsible for generating adequate code from the patterns code templates and the pattern bindings. Every pattern instance is analyzed in its binding context before the generation is started. Therefore, tailored and optimized code can be created.

To generate code for a pattern, not only its own bindings have to be considered, but also other patterns bound to the same target objects. For example, a propagating filter needs information about the pipe to which data changes should be reported. Internal properties (additional bindings) are used to allow the combination of patterns and are used to generate optimized code. Application code is generated by assembling tailored code templates that are part of each pattern. A very simple code fragment may look as follows:

```
'{compute}
   "Does the calculation and stores the result in {result}."
   ^self {result}: (self {calculate}: self {getArguments})'
```

Keywords in brackets ({}) are used as macros. Usually code generation can be done by choosing code templates and replacing all macros with other templates or bound values. More complex patterns (like *Traversal*) also use code synthesis techniques. For further reading see Heister et al. (1997).

5. DISCUSSION

This work combines different software engineering techniques. Structure models are used together with a pattern based design strategy. Application generators are used to implement a simulation component. The approach can

be seen as a domain specific software architecture (DSSA, see Mettala and Graham (eds.), 1992). Domain engineering in the field of building simulation resulted in the overall architecture of a simulation component (*Figure 3*) and in the implementation of the libraries. Also our pattern catalog is domain-specific and part of the domain model. Reference requirements are included in the informal parts of our patterns, prescribing which patterns could be used together or giving hints how certain simulation problems can be solved. To design a simulation component, only the application engineering has to be performed. This includes especially setting up or refining a class diagram for the building structure and instantiating and binding patterns from the catalog. Tool support is given for these tasks. A detailed process of this modeling procedure is not yet defined and will be a topic for future works.

The revised pattern catalog contains behavioral patterns together with patterns describing architectural styles. It is partitioned into categories that deal with different aspects of simulation (the partitioning supports aspect oriented programming (AOP), see Kiczales, 1997). Each category can be seen as a view and be modeled separately. We are currently extending the pattern editor to support views.

The main advantage of the new catalog is that we have found a way to express parts of the component's architecture in (design) models. Patterns can be used to implement or refine an architectural style. The configuration is done by binding a pattern instance to the simulator model. These patterns have a fixed formal interface and code templates (see Heister et al., 1997) and therefore cannot express the whole variety of a style. But as our domain is limited, it is sufficient to use only a few domain-specific implementations of an abstract style.

All our patterns must be able to work together: they form a system of patterns (compare Buschmann et al., 1996). Each individual pattern is used to model a part of the component, but with the right combination of patterns a building simulator can be designed. For example, with our *Pipe* and *Filter* patterns, a data flow between two objects can be defined. At both ends of the pipeline activity may take place. This is usually a calculation of values. The Filter pattern is used to trigger such an activity; the activity itself must be modeled elsewhere (i.e., with patterns from another category). A future topic is to investigate how such dependencies and constraints between patterns can be formally expressed.

6. CONCLUSION

With architectural patterns it is possible to model architectural styles separately. The pattern binding concept of PSiGene allows to combine different pattern instances and to apply them to class diagrams. Therefore, architectural styles can be integrated using a formal interface. The main advantage of the architectural patterns is a better maintainable system (model changes take effect only locally), better tailored components, and the ability to handle more complex models. Also, our pattern catalog became more domain independent. We still have some special simulation patterns but they are able to work together with more abstract and more general architectural patterns. We believe that these architectural patterns can easily be adapted to other domains. Future work will investigate the applicability of our approach in other domains.

Our patterns do not provide the whole bandwidth of all of their possible applications but only a domain-specific subset. This makes code generation and optimization possible but restricts the universal usage of the patterns a bit. Finding more variants and new patterns is also a topic for future works.

A small disadvantage of our new pattern catalog is that it takes more time to model small simulation components as each aspect has to be designed separately. But for larger models this separation of concerns is mandatory and leads to more flexible simulators (e.g., nonfunctional requirements like distribution can be modeled and documented explicitly and more easily).

The pattern-based approach to software architecture seems to be feasible and worked well for PSiGene. Variants of components can be created within short time, and a component can match the architectural demands of an application by changing abstract architectural properties in the models. We therefore believe that our approach to architecture modeling helps the software development in providing and using tailored components.

REFERENCES

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

Bass, L., Clements, P., Kazman, R. (1998) Software Architecture in Practice, *SEI series in software architecture*, Addison-Wesley

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

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stall, M. (1996) *Pattern-oriented Software Architecture - A system of Patterns*. John Wiley & Sons Ltd.

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

Heister, F., Riegel, J. P., Schütze, M., Schulz, S., and Zimmermann, G. (1997) Pattern-Based Code Generation for Well-Defined Application Domains, *European Pattern Languages of Programming Conference (EuroPLoP)*, Siemens Technical Report 120/SW1/FB, 263-273

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. Irwin, J. (1997) Aspect-Oriented Programming, *PARC Technical Report*, February 1997, SLP97-008 P9710042

Kim, J. J., Benner, K. M. (1996) An Experience Using Design Patterns: Lessons Learned and Tool Support, *Theory and Practice of Object Systems*, Vol. 2(1), 61-74

Kruchten, P. B. (1995) The 4+1 View Model of Architecture. *IEEE Software*, 42-50, November 1995

Lieberherr, K. J. (1996) *Adaptive Object-Oriented Software Development: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston

Mettala, E., Graham, M. H., eds. (1992) The Domain-Specific Software Architecture Program, *Special Report CMU/SEI-92-SR-9*, Carnegie Mellon University, Pittsburgh

Monroe, R. T., Kompanek, A., Meltom, R., Garlan, D. (1997) Architectural Styles, Design Patterns, and Objets, *IEEE Software*, January 1997

Schütze, M., Riegel, J. P., and Zimmermann, G. (1997) A Pattern-Based Application Generator for Building Simulation, *European Software Engineering Conference (ESEC)*, Zürich