

# Automated Generation of Monitors for Pattern Contracts

Benjamin Tyler<sup>1</sup>, Jason O. Hallstrom<sup>2</sup>, Neelam Sundarajan<sup>1</sup>

<sup>1</sup>Computer Sc. & Engineering  
Ohio State University  
Columbus, OH 43210, USA

{tyler, neelam}@cse.ohio-state.edu

<sup>2</sup>Computer Science  
Clemson University  
Clemson, SC 29634, USA

jasonoh@cs.clemson.edu

Design patterns help system designers apply proven solutions to commonly occurring problems. While the informal style used to describe patterns is valuable, it is also imprecise. To ensure that patterns are applied correctly, we must also have precise pattern characterizations, and tools for determining whether the appropriate implementation requirements are satisfied.

This paper reports two contributions focused on automating the detection of pattern implementation errors. First, we present a specification language that allows us to capture pattern requirements precisely, as well as the ways in which patterns are specialized in the systems that use them. Second, we present a tool that generates a set of aspect-oriented monitors for a system based on the specifications of the patterns used in its design, and the specifications of the pattern specialization details. The generated aspects are used to monitor the system at runtime to determine whether the appropriate implementation requirements are satisfied. The aspects are valuable across the development lifecycle, where they aid in maintaining the design integrity of the system by detecting errors introduced during evolution and maintenance. We demonstrate the specification and monitoring approach with a case study.

## 1. INTRODUCTION

*Design patterns* [6, 3, 19, 17] have fundamentally altered the way system designers approach the design of large software systems. Their benefits are two-fold. First, patterns make it possible for designers to exploit the collective wisdom and experience of the software community. Second, they provide an extended design vocabulary, allowing designers to more quickly understand the structure of a given system, and to gain a deeper understanding of why it behaves in particular ways. As Buschmann *et al.* [3] put it, “patterns support the construction of software with *defined properties* [emphasis added]”. To fully realize these benefits, however, two requirements must be addressed. First, patterns must be described *precisely*. Each description must specify both the implementation requirements that must be satisfied if the pattern is to be used as intended, as well as the system behaviors that are guaranteed by virtue of the pattern’s application — the ‘defined properties’ discussed by Buschmann *et al.* Second, designers must have suitable

tools for automating the detection of system implementation errors based on the specifications of the patterns used in its design. These requirements are especially important when patterns are applied in safety- and mission-critical systems.

In this paper, we report two contributions that address the aforementioned requirements. The first contribution is a pattern specification language designed to support runtime specification checking. We refer to this language as the *Pattern Contract Language (PCL)*, pronounced ‘pickle’. Given a pattern  $P$ , the PCL *contract* for  $P$  will specify the requirements that must be satisfied by any system in which the pattern is used, and the resulting system behaviors that are guaranteed as a result. Note, however, that patterns can be tailored to many different system contexts. These *specialization* details fundamentally affect the precise implementation requirements, as well as the behaviors that should be expected. In PCL, this information is captured in the form of a *subcontract*. Hence, the contract for  $P$  specifies information common to all applications of the pattern, and a particular subcontract specifies how the pattern is specialized for use in a given system.

The second contribution is the design and implementation of *MonGen*, a tool for automating the creation of runtime *contract monitors*. Given a set of PCL contracts and subcontracts for a particular system, MonGen automatically generates a set of *monitoring aspects* in AspectJ [10]. The generated aspects are used to monitor the system during its execution to determine whether the requirements and behavioral guarantees specified in the pattern contracts and subcontracts are met. These aspects are valuable not only during the initial stages of the development process, but *throughout* a system’s lifecycle. By reusing the generated aspects during periods of evolution and maintenance, the monitoring approach can detect errors that would otherwise violate the *design integrity* of the system. We will return to this point in Section 7.

In designing MonGen, we chose an aspect-oriented approach for two reasons. First, using AspectJ allows us to defer the task of weaving monitoring code throughout the system to the AspectJ compiler. Second, and more important, many design patterns address issues involving multiple classes. Consequently, the contracts for these patterns specify requirements that must be monitored across the system hierarchy. Aspect-oriented languages such as AspectJ provide convenient language constructs for capturing this type of *cross-cutting* concern.

One of the important motivations behind the development of PCL as a language in which patterns can be specified for-

mally and precisely is that standard informal descriptions of many patterns are inherently ambiguous. Consider, for example, the standard informal description of the Observer pattern [6]. This description elucidates a number of key properties. It is clear, for example, that objects participating in the Observer pattern play one of two roles: Subject<sup>1</sup> or Observer. The intent of the pattern is to keep multiple observers *consistent* with the state of a single subject. When an object wishes to become an observer of a subject, it invokes the subject’s `Attach()` method. When it is no longer interested, it invokes the `Detach()` method. In addition, the subject provides a `Notify()` method that invokes an `Update()` method on each attached observer. The informal description states that the ‘... subject notifies its observers [by calling `Notify()`] whenever a change occurs that could make its observers’ states inconsistent with its own’. But how will the subject determine whether a change in its state has caused it to become inconsistent with one or more observers? Indeed, what does it mean to say that the subject’s state is *inconsistent* with an observer? These are the types of ambiguities that can lead to software defects if members of a design team have subtly different interpretations of the pattern’s description. Our pattern specifications provide precise answers to these questions, and our monitoring tools detect implementation inconsistencies before they reach system deployment.

Note, however, that precision comes at the risk of reduced flexibility [16]. If, for example, we were to adopt a single definition for the notion of *inconsistency* in the Observer pattern, the pattern would not be applicable in systems that have a different notion of this concept. Preserving flexibility is one of our key concerns, and PCL is designed to avoid such a compromise. We achieve this, in part, by parameterizing our specifications using *auxiliary concepts*. Each concept is a relation involving one or more states of the objects participating in the pattern. Pattern contracts are expressed in terms of these concepts, but the actual definitions are supplied as part of the subcontracts corresponding to particular specializations of the pattern. Consider, for example, the contract for the Observer pattern. This contract will declare two auxiliary concepts: *Modified()* and *Consistent()*. The first concept will capture whether the state of the subject has been modified in a way that could result in inconsistency with an attached observer. The second will capture whether the state of the subject is consistent with the state of a particular observer. By expressing the contract over these relations, but deferring their definitions to a subcontract, we are able to achieve precision without a compromise in flexibility.

Several authors have considered how to formalize design patterns. We will discuss a number of proposed approaches in Section 6. At this point, however, we will note that our work is unique in two important ways. First, although other authors have considered pattern formalization, our work is the first to propose a specification language that precisely captures the implementation requirements and behavioral guarantees associated with a range of patterns, while simul-

taneously accommodating the variation that occurs across applications of the same pattern. Second, our work is the first to consider automating the generation of monitoring code for detecting pattern implementation errors based on the relevant pattern specifications.

We should also mention that in [20], we consider the formalization of design patterns, and in [21], we consider runtime monitoring of pattern contracts, coding the necessary monitors by hand. While there are similarities between the work presented here and the work presented in our previous papers, there are a number of important differences. First, we have not previously considered a general purpose pattern specification language, such as PCL. Second, we have not previously considered pattern specializations, and by consequence, we have not considered subcontracts. Finally, we have not previously considered automating the generation of contract monitoring code. Despite these differences, however, the results reported in this paper build upon important principles introduced in [20] and [21].

In Section 2, we present the main details of PCL. In Section 3, we outline how MonGen generates aspects and subaspects from pattern contracts and subcontracts. In Section 4, we develop the contract for Observer, briefly outline a simple system built using the pattern and the corresponding pattern subcontract, and consider some of the code MonGen generates from them. In Section 5, we summarize our experiences using the approach with other case studies. In Section 6, we discuss related work. In Section 7, we conclude with a summary of our approach and explain how it is applicable across the full software lifecycle.

## 2. PATTERN CONTRACT LANGUAGE

A design pattern consists of a number of *roles*. One set of requirements that the pattern imposes is that these roles interact in specific ways. In the Observer pattern, one such requirement, as we saw, is that when the state of the Subject changes and its new state might be *inconsistent* with that of the Observers, it should invoke the `Update()` operation on each observer. In our formalism, these requirements will be part of the *role contract* corresponding to the Subject role. In general, the *pattern contract* for a given pattern will include a role contract corresponding to each role of the pattern. The pattern contract will also specify a *pattern invariant*, an assertion involving the states of all the roles. The invariant will be satisfied whenever control is outside all of the methods of all the roles. The invariant is, essentially, the ‘defined properties’ [3] that the correct use of this pattern ensures for the system. The role contracts, as well as the invariant, will be expressed using *auxiliary concepts*. As noted earlier, however, the definitions of the concepts will not be part of the pattern contract, but will instead be provided in the subcontracts.

Although the definitions of the concepts in the subcontracts may be tailored in whatever way is necessary to meet the needs of a particular application, it often turns out that if a pattern contract involves more than one concept, then these concepts’ definitions in any application must satisfy certain *constraints*, else the intent of the pattern will be violated even if the system meets all other requirements of the pattern. We will see this in our contract for the Observer pattern. Thus an important part of the pattern contract will be the constraints that the various auxiliary concepts used in the contract must satisfy.

<sup>1</sup>We use names starting with uppercase letters, such as Subject, for roles. We use corresponding lowercase names, such as subject, to refer to the objects that play these roles. In some cases the name of a pattern is also used for one of the constituent roles, as in the case of the Observer role of the Observer pattern. In such cases, the context will make clear which is intended.

A pattern contract must specify two additional items. First, it must specify the *instantiation* clause. Second, it must specify the *enrollment* clause for each role of the pattern. The instantiation clause will specify how a new instance of the pattern will be created. We call any group of objects interacting according to a given pattern an *instance* of the pattern. For example, in a particular system built using the Observer pattern, at any given point during execution, we may have several groups of objects, with each group consisting of one object playing the Subject role, and the other objects playing the Observer role and “observing” the first object. Each such group is an instance of the Observer pattern. The instantiation clause will specify how a new instance of the pattern is created. Note that the group of objects in a given instance is not static. In the Observer pattern, a new object may wish to start observing an object that is already being observed by several other objects. This object may do so by invoking the Attach() method on the subject. This will be specified, in our contract for the pattern, as the enrollment clause for this role.

The grammar<sup>2</sup> for pattern contracts appears in Fig. 1. A pattern contract consists of the name of the pattern ( $\langle pid \rangle$ ), the auxiliary concepts needed, the constraints that must be

```

<patternContract> ::= pattern <pid> contract {
    <auxConcepts> <constraints>
    <instantiation> <invariant>
    <roleContracts> }
<auxConcept> ::= concept <cid> ( <rids> )
<constraint> ::= ... predicate on auxiliary concepts
<instantiation> ::= instantiation: <rid>.<mid> { <cond> }
<invariant> ::= ... assertion involving various roles
    and auxiliary concepts
<roleContract> ::= role <rid> contract {
    <roleStateSpecs> <enrollment>
    <namedMethodSpecs> <othersSpec> }
<roleStateSpec> ::= state { ... role state ... }
<enrollment> ::= enrollment: <rid1>.<mid> { <cond> }
<namedMethodSpec>
    ::= ... standard method specification with
        requires, preserves, ensures clauses
<othersSpec> ::= ... only preserves, ensures clauses. . .

```

Fig. 1. Partial Grammar of PCL (part 1)

satisfied by the definitions of the auxiliary concepts, the pattern invariant, the instantiation clause, and the contract for each role.

Corresponding to each auxiliary concept, we specify its name ( $\langle cid \rangle$ ), and the list of role names that the concept is concerned with. We considered the possibility of allowing *default* definitions for some or all the concepts in the pattern contract, but decided against it because of the risk that a needed definition might be omitted from a subcontract by mistake. Another alternative would be to allow default definitions, but to have MonGen print a warning whenever a default is used. Next we have the set of constraints that must be satisfied by the auxiliary concepts (as defined in any subcontract of this contract).

The instantiation clause specifies the particular method ( $\langle mid \rangle$ ) of the particular role ( $\langle rid \rangle$ ) that must be invoked

<sup>2</sup>Due to space limitations, we omit many details and present a simplified version of *PCL*. A full version is available in [22].

to create a new instance of the pattern. In some cases, a new instance of the pattern is created as soon as an object that will play the particular role is constructed, rather than at a later point when a method is invoked on that object. In such a case,  $\langle mid \rangle$  will be specified as the ‘method’ new, representing a call to the appropriate constructor. The  $\langle cond \rangle$  in this clause specifies a condition that must be satisfied at the time of the instantiation. If, during execution, at the time of the call to this method (or constructor), the specified condition is not satisfied, the monitoring code generated by MonGen will print a warning, and no new instance of the pattern will be created<sup>3</sup>. Only one instantiation clause appears in a pattern contract; multiple clauses could have been allowed but in the patterns we have studied, this generality was not needed.

The object involved in the instantiation, i.e., the one playing the role  $\langle rid \rangle$  in the instantiation clause, will be called the *lead object* of this instance. Since, as we noted above, at any point during execution of a given system built using a particular pattern, we may have several different instances of the pattern, we need a suitable way to refer to these different instances and the lead object will serve this purpose. Thus in the contract for the Observer pattern in Section 4, the instantiation clause will specify that a new instance is created when a new object playing the Subject role is constructed. This object will be the lead object for this instance. In the aspect that MonGen generates from a pattern contract, information about all the instances of a pattern that exist at a given point during execution are maintained in a single data object. The code in the aspect uses the lead object as an index into this data object to access information about the corresponding instance.

Let us now consider the *role contract* corresponding to the role  $\langle rid \rangle$ . First, we specify the *role state* of the role. This will list the name of each variable of the role and its type in the standard variable declaration format. For some of these variables, their type will be one of the role names of this pattern. Next we have the enrollment clause for this role. This, as in the case of the instantiation clause, is a call to a particular method (or constructor). The enrollment clause must identify the enrolling object and the lead object of the instance in question since the latter will identify the particular instance that the object is enrolling in. We have omitted the syntax for this. Following the enrollment, the specified condition must be satisfied, else the monitoring code generated by MonGen will produce a warning at runtime.

Note that the instantiation clause is also an enrollment clause since, at the time of instantiation, the lead object is enrolling in the new pattern instance. If additional objects can enroll in the same role in this pattern instance, they will do so by performing the action specified in the enrollment clause for this role. Or, if additional objects are not allowed to enroll in this role—as is the case with the Subject role of Observer in which each instance of the pattern has only one subject enrolled—then the enrollment clause will be missing from the role contract for this role.

Following the enrollment clause, we have specifications of

<sup>3</sup>Note that if this condition is not satisfied, that doesn’t necessarily indicate that there is a bug in the system. It may be that this method is used for more than one purpose and in some situations, it is not the intent of the designers to use this pattern. Nevertheless, it seems prudent to provide a warning in this situation.

the “*named*” methods and the “*other*” methods. The named methods are the ones that this role must have in order to play its part in the pattern. For example, the Observer role of the Observer pattern must provide an Update() method; that is one of the named methods of this role. Suppose, in a specialization of a given pattern  $P$ , the class  $C$  plays the role  $R$ . The methods of  $C$  that correspond to the named methods of  $R$  will be required to satisfy the corresponding specifications in  $R$ ’s role contract. But, in general,  $C$  will provide *additional* methods; these are its *other* methods. If these methods are not suitably designed, then the intent of the pattern may be compromised. For example, the role state for Subject will specify that this state must include a variable whose value will be a set of references to the objects enrolled to observe the subject. Now if a class playing the Subject role were to include a method that changed or destroyed the information in this variable, the system would clearly fail, even if all the methods explicitly listed in the Subject role were implemented correctly. The *others* specification imposes conditions to prevent such problems, and these must be met by all these methods. These specifications will only involve the *ensures* and *preserves* clauses since the methods’ pre-conditions cannot in any way interfere with what the named methods of the role do.

The grammar for *subcontracts* appears in Fig. 2. A subcontract lists the name of the specialization, the name of the pattern it specializes, provides a set of role maps (one corresponding to each role of the pattern), and a set of definitions for the auxiliary concepts of the pattern. A role map specifies how a particular class (or interface, but we only consider classes)  $\langle cid \rangle$  plays the part of a particular role  $\langle rid \rangle$  of the pattern. The *state map* and *interface map* together tell us

```

(subContract) ::= subpattern  $\langle sid \rangle$  specializes  $\langle pid \rangle$  {
                 $\langle roleMaps \rangle$   $\langle auxConceptDefs \rangle$  }
(roleMap)     ::= rolemap  $\langle cid \rangle$  as  $\langle rid \rangle$  {
                 $\langle stateMap \rangle$   $\langle interfaceMap \rangle$  }
(stateMap)   ::= state: {  $\langle varMaps \rangle$  }
(varMap)     ::=  $\langle rvid \rangle = \{ \dots \text{code} \dots \}$ 
(interfaceMap) ::= methods: {  $\langle methodMaps \rangle$  }
(methodMap)  ::=  $\langle rmid \rangle(\langle rmargs \rangle)$ : {  $\dots$  details omitted }
(auxConceptDef) ::= auxiliary concept
                   $\langle auxid \rangle$  ( $\langle cargs \rangle$ ) {  $\dots$  code  $\dots$  }

```

Fig. 2. Partial Grammar of PCL (part 2)

exactly how we can think of objects that are instances of this class as instances of this role. The *state map* is a set of *variable maps*, one corresponding to each variable listed in the role contract for this role in the pattern contract. A variable map lists the name of the particular variable defined in the role ( $\langle rvid \rangle$ ), and the *code* that takes the current state of the  $\langle cid \rangle$  object and returns the value of this particular role variable when the  $\langle cid \rangle$  object is viewed as an  $\langle rid \rangle$  object. Thus the variables listed in the role contract for  $\langle rid \rangle$  in the pattern contract do not have to be part of  $\langle cid \rangle$ . Instead, given any state of the  $\langle cid \rangle$  object, the code listed in the  $\langle varMap \rangle$  corresponding to any variable of the  $\langle rid \rangle$  role will give us the value of that variable when we view the  $\langle cid \rangle$  object in this role. In simple cases, each role variable will correspond to a variable in the class, and the code will simply return the value of that variable; but *PCL* allows for more complex maps. The *method maps* in the *interface map* similarly define the mappings between the methods of the

class and the role. Again in simple cases, particular methods of the class will map to each method of the role. We omit details of method maps.

Following the role maps, we have the auxiliary concept definitions. For each auxiliary concept  $\langle auxid \rangle$ , and each possible combination of classes that play the various roles that appear as its parameters, we must provide a definition. This is done by providing suitable code that performs appropriate comparisons of the objects of the particular classes mapped to each role, and returns a *true* or *false* value to indicate if the relation represented by the concept is satisfied or not. An alternative approach would have been to require the concept definitions to be expressed as mathematical expressions. But having code here, as well as in the  $\langle varMap \rangle$ s, makes the job of MonGen simpler.

Before concluding this section, we briefly consider some important notations and mechanisms that are included in the formalism in order to enable us to express, in the pattern contracts, certain common requirements. First, we often need to refer to the object playing the lead role in a pattern instance. For this purpose, the formalism allows the use of the keyword “lead” in specifications. Similarly, the keyword *players* may be used to refer to the *vector* of all the objects enrolled in a given instance, in the order they enrolled.  $\text{players}[k]$ , the  $k^{\text{th}}$  entry in this vector, will consist of a reference to the particular object and the name of the role it enrolled in. The aspect that MonGen creates for a given pattern contract will, when the system being monitored performs the action specified in the instantiation clause of the pattern contract, create a new data object representing the  $\text{players}[]$  for this new instance, and initialize it to contain information about the lead object. When the action specified by the enrollment clause of any role is performed, it will add the enrolling object to the appropriate  $\text{players}[]$  variable. PCL also allows individual role contracts to include a *disenrollment* clause which we do not show in our grammar. When the action specified in this clause is performed by the monitored system, the aspect code will remove the corresponding object from the appropriate  $\text{players}[]$  object. MonGen also recognizes the keyword *lead* and produces appropriate code corresponding to it.

Many patterns require specific methods to be invoked in a specific order under various conditions. Thus in *Chain of Responsibility*, when one of the objects in the chain receives a call that it cannot handle, it must then invoke an appropriate method on the next object in the chain, which must in turn act in a similar manner. In PCL, such requirements are expressed in terms of method *call sequences* or *traces*. The specification of a method  $m()$  of a role  $R$  will, in general, specify the effect of  $m()$  on the variables of  $R$ , as well as the sequence of calls that  $m()$  makes during its execution. Each call made by  $m()$  will be represented as an element in  $\tau.m$ , the trace associated with  $m()$ . This element will record the name of the method invoked, the identity of the object on which the method is invoked, and the other arguments for the invocation. The aspect produced by MonGen from a given pattern contract includes a trace corresponding to each method of each role. As the monitored system code executes, the aspect intercepts calls made by various methods to record appropriate information in the appropriate trace about the particular call. This will allow the aspect, when a method  $m()$  finishes execution, to check whether the requirements contained in  $m()$ ’s specification in

the corresponding role contract, including the requirements concerning the calls that  $m()$  made during execution, are indeed satisfied – and if not, to print appropriate warnings. PCL also includes a number of useful auxiliary functions and predicates on traces, as well as on the `players[]` variable, to simplify expression of common requirements. MonGen contains implementations of these auxiliary functions<sup>4</sup>.

### 3. MONITOR GENERATION

Given contracts for particular patterns and subcontracts corresponding to how they are specialized in a given system  $S$ , *MonGen* automatically generates aspects and subaspects in *AspectJ* [10]. These can be *woven together* with  $S$  by the *AspectJ* compiler to produce a system that, when executed, will produce the same results as  $S$ , but will *monitor*  $S$  as it executes, and print appropriate warning messages when any portion of any pattern contracts, specialized as specified in the subcontracts, are violated during execution.

We start with a summary of the main parts of *AspectJ* that we use. A *join point* identifies particular places in the execution of a program. The only type of join point we use is method execution. A *pointcut* groups together a set of join points that we want to treat in a uniform manner. Pointcuts enable us to collect *context*, for example, the object on which the method in question was applied, parameter values, etc. Finally, the *advice* associated with a pointcut specifies the code that should be executed when control reaches any of the join points that match the pointcut. We use two kinds of advice. The *before* advice, if any, is executed before the method corresponding to the particular join point is executed; the *after* advice after the method is executed. Like classes, aspects can also contain their own local state and methods; and the various advice code that MonGen produces will manipulate the local state of the aspect.

#### 3.1 Aspects from Pattern Contracts

Given a pattern contract, for each *named* method of each role, a pointcut as well as a block of *before* and *after* advice associated with that pointcut are created. The *before* advice checks the precondition specified in the contract for the particular named method; the *after* advice checks the postcondition. Since the actual methods of the class(es) playing the particular role corresponding to the particular named method will be specified in the subcontract, the pointcut will, in the aspect corresponding to the pattern contract, be labeled *abstract*. The subaspect generated from the subcontract will provide the definition for the pointcuts corresponding to each named method. Similarly, the functions corresponding to the various auxiliary concepts will be labeled as *abstract* in the aspect with their definitions being provided in the subaspects. The pointcut corresponding to the *other* methods can simply be defined as any method execution *not* included in the other pointcuts (the ones corresponding to the named methods).

The variables in the role states present a challenge. As we saw in Section 2, these variables may not actually be part of the classes playing the particular roles. At the same time, unlike for pointcuts or methods, we cannot label variables as

<sup>4</sup>We should note that having a single `players[]` vector to hold information about all the enrolled objects does raise some type issues since, in general, these objects will be enrolled to play different roles. In this paper, we ignore these issues.

```

<patternContract> ::= pattern <pid> contract {
  <auxConcepts> <constraints> <instantiation>
  <invariant> <roleContracts> }
MonitorAspect(<patternContract>) ←
  “abstract aspect ” + Text(<pid>) + “_monitor{”
  + ConceptAbsMeths(<auxConcepts>)
  + AbsPCs(<roleContracts>) + OtherPCs(<roleContracts>)
  + Advices(<roleContracts>)
  + InvCheckCode(<invariant>)
  + ... additional advice code omitted... + “}”
... other attributes omitted ...

```

Fig. 3. PCL Attribute Grammar (partial)

*abstract*. To handle this, the aspect that MonGen generates from the pattern contract includes an abstract method corresponding to each variable in each role state. These methods will then be defined in the subaspect by the code provided as part of the corresponding  $\langle varMap \rangle$ s in the subcontract.

The value of *MonitorAspect()*, the main synthesized attribute of  $\langle patternContract \rangle$ , as defined in Fig. 3, will be the abstract aspect produced by MonGen corresponding to the given pattern contract. It is obtained by appending together (using ‘+’, *Java*’s append operator) the abstract methods corresponding to the auxiliary concepts; the abstract pointcuts corresponding to the various named methods in the various roles; the pointcuts corresponding to the *other* methods for each role; the advice code which will, on the one hand, perform various bookkeeping operations including updating appropriate trace variables, and, on the other hand, check that preconditions (for named methods) and postconditions (for named as well as other methods) are satisfied at the appropriate points; and the advice code for checking that the pattern invariant is satisfied at appropriate points. Note that the advice code corresponding to the  $\langle instantiation \rangle$  clause does not appear explicitly in the construction of *MonitorAspect()*. Instead, information about that clause is passed down (using inherited attributes not shown in Fig. 3) to the role contracts and used in the construction of the advice code corresponding to the role that appears in the instantiation clause.

```

<roleContract> ::= role <rid> contract {
  <roleStateSpecs> <enrollment>
  <namedMethodSpecs> <othersSpec> }
AbsPCs(<roleContract>) ← AbsPCs(<namedMethSpecs>)
OtherPC(<roleContract>) ← NegNamedPCs(<namedMethSpecs>)
Advice(<roleContract>) ←
  NMethAdvice(<namedMethSpecs>) + OthersAdv(<othersSpec>)

```

Fig. 4. PCL Attribute Grammar (contd.)

Fig. 4 shows part of the attribute grammar rules for  $\langle roleContract \rangle$ . The (abstract) pointcuts corresponding to the named methods are obtained based on their specifications. Note that the only information required is the *names* of the methods, not their pre-/postconditions. The pointcut corresponding to the other methods is obtained as the *complement* (using ‘!’, the negation operation of *AspectJ*) of the pointcuts corresponding to the named methods. The actual advice that checks that the pre- and postconditions in the case of the named methods and the postcondition in the case of the other methods are all satisfied at the appropriate points is obtained by combining the advice corresponding to the respective specifications.

### 3.2 Subaspects from Subcontracts

Let us now turn to how, given the subcontract corresponding to how a pattern  $P$  is specialized in a particular system, MonGen generates the corresponding subaspect. The subaspect will provide definitions for the abstract pointcuts as well as the abstract methods in the aspect that *MonGen* produced from  $P$ 's contract.

```

⟨subContract⟩ ::= subpattern ⟨sid⟩ specializes ⟨pid⟩ {
  ⟨roleMaps⟩ ⟨auxConceptDefs⟩ }
MonitorSubaspect(⟨subContract⟩) ← “aspect ” + Text(⟨sid⟩)
+ “_monitor extends ” + Text(⟨pid⟩) + “_monitor{”
+ ParentClauses(⟨roleMaps⟩)
+ RSMETHODS(⟨roleMaps⟩)
+ CONCPCs(⟨roleMaps⟩)
+ ConceptDefns(⟨auxConceptDefs⟩) + “}”

```

Fig. 5. Grammar Rule for  $\langle subContract \rangle$

In Fig. 5, the value of the *MonitorSubaspect()* attribute is the subaspect for the given subcontract. The *ParentClauses()* attribute is essentially the code that defines each of the classes that plays a given role as implementing the corresponding interface. *RSMETHODS()* defines the methods corresponding to each of the variables in the *role states* of the various roles. Recall that in defining the  $\langle varMaps \rangle$  that are part of the  $\langle roleMap \rangle$  corresponding to a particular class  $C$  playing a particular role  $R$ , we are required to provide code that, given the current state of an object that is an instance of  $C$ , gives us the value of the particular role variable when the  $C$ -object is *viewed as an  $R$ -object*. The attribute evaluation rules for *RSMETHODS()* use this code directly. The *ConceptDefns()* similarly uses the code in the  $\langle auxConceptDefs \rangle$  that provide the definitions of the auxiliary concepts for this particular specialization of the pattern. The value of *CONCPCs()* is the set of definitions for the pointcuts corresponding to the named methods of the role; this is obtained in a straightforward manner from the  $\langle methodMaps \rangle$  in the subcontract.

Some additional points are worth noting. First, MonGen includes, in the aspect for a given pattern contract, an empty interface  $R$  corresponding to each role  $R$  of the pattern. The classes that, according to the subcontract, play the role  $R$  will, in the corresponding subaspect, be specified to implement the interface. Second, we have not considered how exactly the code for checking various assertions, in particular, the invariant, the pre- and postconditions for the named methods and the postconditions for the other methods, is generated. Recall that these assertions are expressed in terms of the role variables. Essentially, the advice code that checks these assertions at the appropriate points, invokes the abstract methods corresponding to these role variables. The methods themselves, as we saw, are defined based on the code in the  $\langle roleMaps \rangle$  in the subcontracts. The advice code corresponding to these assertions uses the values returned by these methods to check that the assertions are (or are not) satisfied at various points, and prints warning messages as appropriate. Second, we have not considered how the  $\langle enrollment \rangle$  clause is handled. The aspect corresponding to this not only has to check, when the corresponding method finishes, that the specified condition is satisfied, but also take care of additional bookkeeping by updating `players[]` to add the newly enrolled object.

One other point should be noted. The constraints that must be satisfied by auxiliary concept definitions provided

in the subcontract pose a problem. They typically involve universal quantifiers. Thus the constraint in the contract for the Observer pattern in Section 4 is universally quantified over two subject states and an observer state. But there is no way for the aspect and subaspects that MonGen generates to check whether such a constraint is satisfied. One possibility would be to check that the constraint is satisfied for the states that actually arise during execution; we are exploring this in our current work.

### 3.3 Pattern Instances and Method Traces

Since several instances of a pattern  $P$  may simultaneously exist at runtime during the execution of a system  $S$ , *MonGen* stores information about all of them. The aspect generated from  $P$ 's contract contains a Set of *PatternInstance* objects, each labeled with the lead object of that instance. The advice corresponding to the instantiation clause adds a new element to this set, corresponding to the newly created instance. Each *PatternInstance* object contains references to all the objects playing roles in that instance. A related point is that a given object may participate in multiple instances of the same pattern. For example, in the case of the Hospital system presented in Section 4, a given nurse object may be observer for several patient objects. To deal with this, the *after* advice code generated by MonGen corresponding to method calls on objects that play the observer role searches through all pattern instances to locate the ones in which the given object (on which the particular method was applied) is enrolled; and checks that the corresponding assertions are satisfied.

*MonGen* also has to save the trace for each method as it is executed, since both *named* as well as *other* method specifications may refer to them. The *before* advice for each method  $m()$  of each role creates and initializes an empty trace; this is used to record calls that  $m()$  makes. When  $m()$  finishes execution, its *after* advice uses this trace to check that the conditions listed in  $m()$ 's postcondition are satisfied. Consider a call that a method  $n()$  makes to the method  $m()$ . The *before* advice for  $m()$  not only creates and initializes the new trace for  $m()$ , it also updates the trace of  $n()$  to record  $n()$ 's call to  $m()$ . Similarly, the *after* advice for  $m()$  also records suitable information on the trace of  $n()$  about the results returned by  $m()$ . There are a number of additional details concerning MonGen that we omit for lack of space. MonGen was developed using the *ANTLR* parser generator [14], and is available at [22].

## 4. CASE STUDY

We develop (part of) the contract for the Observer pattern. Then we present some details of a simple system built using the pattern, and develop (part of) the corresponding subcontract. Along the way, we present some fragments of the code generated by MonGen.

The contract for the Observer pattern uses two auxiliary concepts: *Consistent*, which represents the notion of whether a given subject state is consistent with a given observer state; and *Modified*, which represents what it means for a given subject state to be *sufficiently different* from another state of the subject (to require notification of the observers). The constraint may be understood as follows: Suppose the current state of the subject is `s1`, the current state of an observer is `o1`, and `s1` is, according to the definition of *Consistent()* (to be provided in the subcontract), consistent

```

pattern Observer contract {
  concept Consistent(Subject, Observer);
  concept Modified(Subject, Subject);
  constraint:
     $\forall s1, s2, o1 :$ 
       $[ \neg \textit{Modified}(s1, s2) \wedge \textit{Consistent}(s1, o1) ]$ 
       $\Rightarrow [ \textit{Consistent}(s2, o1) ]$ 
  instantiation: new Subject() {
    (newOb._observers =  $\Phi$ ) }
  invariant:
    Subject(players[0])  $\wedge$  Observer(players[1 :])  $\wedge$ 
    ( $\forall ob : (ob \in \text{players}[1:].\text{objs}) ::$ 
      Consistent(players.objv[0], ob))

```

Fig. 6. Observer Contract (part 1)

with  $o1$ . Suppose the subject state changes to  $s2$  and that  $s2$  is, according to the definition of *Modified*(), not modified from  $s1$ . The constraint requires that in this case,  $s2$  must be consistent with  $o1$ . Otherwise, we will have a problem since the subject will not invoke *Update*() on its observers (since its state has not ‘modified’), but the state of the observer in question will no longer be ‘consistent’ with that of the subject. The problem arises not because of the failure of the subject or the observers to interact in the manner intended by the pattern, but rather having mutually incompatible notions of, on the one hand, *consistency* between the state of a subject and an observer, and, on the other hand, what it means for the state of the subject to be materially *modified*. The constraint prevents such incompatibilities. But, as noted in Section 3, MonGen cannot check, for given subcontracts, whether this constraint is violated.

The *instantiation clause* specifies that a new instance of observer is created when an instance of the (class playing the) Subject role is constructed. Following this construction, the value of the `_observers` field of the subject is required to be empty. This represents the fact that at this point, no observers have enrolled to observe the subject.

The invariant states that the first object to enroll will play the Subject role (as already stated by the instantiation clause), and that all other enrolling objects will play the Observer role. Note that in the second clause, `players[1:]` denotes all the elements of `players[]` starting from its second element (the one indexed 1). The essence of the Observer pattern is that the states of the observers will be *Consistent* with the state of the subject whenever none of the objects are being acted upon by any of the methods of their classes. This is captured by the last clause of the invariant.

The Subject role contract states that this role’s state consists of a single variable, `_observers`, whose value will be the set of references to all the objects currently observing subject. The specification of *Attach*() states that it may be called only if `ob` is not already enrolled in the pattern instance. In the *ensures* clauses, we use the “#” notation to denote the value that the variable had when the method started execution. We also use “ $|\tau|$ ” to denote the length of the particular trace, i.e., the number of calls recorded in it. Finally, we use a “dot” notation to extract particular elements from a given trace, for example the elements involving a particular object, or involving calls to a particular method on a particular object, etc.

The *ensures* clause of *Attach*() states that the enrolling observer is added to `_observers`. The next clause requires that the state of the subject not be *Modified*. This may seem strange since we just stated that the `_observers` field must be

```

role Subject contract {
  state { Set _observers; }
  void Attach(Observer ob):
    requires: (ob  $\notin$  _observers)
    ensures: [(_observers = #_observers  $\cup$  {ob})
       $\wedge$   $\neg \textit{Modified}(\#this, this)$ 
       $\wedge (|\tau| = 1) \wedge (|\tau.\text{ob}.\text{Update}| = 1)$ ]
  void Notify( ):
    preserves: _observers
    ensures: [ $\neg \textit{Modified}(\#this, this)$ 
       $\wedge (|\tau| = |\_observers|)$ 
       $\wedge \forall ob \in \_observers : (|\tau.\text{ob}.\text{Update}| = 1)$ ]
  others:
    preserves: _observers
    ensures: [ $\neg \textit{Modified}(\#this, this) \wedge (|\tau| = 0)$ 
       $\vee ((|\tau| = 1) \wedge (|\tau.\text{this}.\text{Notify}| = 1))$ ]

```

Fig. 7. Subject role contract (partial)

modified! The point is that this change has to do with how the subject keeps track of which objects are observing it, and is not the kind of change the observers would be interested in. Hence, this kind of change should be transparent, so to speak, as far as the *Modified*() concept is concerned. Indeed, definitions of auxiliary concepts commonly ignore the ‘pattern-portion’ of the states of participating objects.

The condition captured by the last two clauses of this *ensures* is often overlooked in informal descriptions of the pattern. The *Attach*() method adds a new observer. Therefore, we must invoke its *Update*() method, else it may not be consistent with the subject. The first of these clauses states that the length of the method call trace  $\tau$  is 1, i.e., there is one method call recorded on it. The second clause states that this call is to the method *Update*(), the method being invoked on the attaching object.

The specification of *Notify*() requires that `_observers` not be changed, that *Notify*() not *modify* the state of the subject, and that *Update*() be invoked on each attached observer. The *others* specification requires that other methods (of the class playing this role) must preserve the `_observers` set. Further, they must either not modify the subject state, or must invoke the *Notify*() method, which, as we just saw, will in turn invoke *Update*() on each attached observer.

```

role Observer contract {
  state { Subject _subject; }
  void Update():
    preserves: _subject
    ensures: (|\tau| = 0)  $\wedge$  Consistent(_subject, this)
  others:
    preserves: _subject
    ensures: Consistent(_subject, #this)
       $\Rightarrow$  Consistent(_subject, this)

```

Fig. 8. Observer role contract (partial)

The Observer role contract lists `_subject` as the only role state variable. The specification of *Update*() requires it to preserve `_subject` (so the reference to the subject being observed is not lost), not invoke any methods, and make the state of the observer consistent with that of the subject. The *others* specification requires `_subject` to remain unchanged (again so that the reference to the subject being observed is not lost). Further, if the state of the observer at the start of the method was *consistent* with its `_subject`’s state, then

the state of the observer at the end of the method must be *consistent* with the `_subject` state. Interestingly, this allows an `others` method of this role to modify the state of the observer – as long as the modification does not affect the *consistency* of this state with that of the subject state. Standard descriptions of the pattern suggest that the state of an observer should *not* change except when the `Update()` method is invoked, which is unnecessarily restrictive. This illustrates how the process of developing the pattern contract in our formalism points to dimensions of flexibility that may be missing in the standard informal descriptions.

```

after(Observer _this): Update_PC(_this) {
  //get #this from caller's trace record
  assert(_subject(#this)==_subject(this));
  //check my trace to see if length = 0;
  assert(Consistent(_subject(this), this));
  //update caller's trace record;
}

```

Fig. 9. after advice generated for `Update()` (partial)

Fig 9 shows the skeleton of the after advice generated by MonGen for `Update()`, with some of the code replaced by comments since the actual code produced by MonGen involves various internal details. In effect, this code gets `#this`, the state the current observer was in at the start of the method from the trace of the calling method. Next, it checks that `Update()`'s *preserves* clause is satisfied; the next two lines check the conditions in the *ensures* clause, and the last line updates the caller's trace record. Note the calls to the method `_subject()`. This will be defined based on the subcontract, in particular the *(varMap)* that specifies how the member variables of the particular class that plays the Observer role are mapped to the `_subject` of the role. Similarly, the call to `Consistent()` will be to the definition provided in the subcontract for this concept. We should stress that in Fig. 9 we have left out many important details that the actual code produced by MonGen accounts for, including, in particular, dealing with multiple instances of the pattern. Another key detail is checking that the pattern invariant is satisfied at the end, provided control is not inside any of the methods involved in the pattern.

Next we briefly discuss Hospital, a simple system that uses the Observer pattern. There are three main classes: Patient, which plays the Subject role, and Nurse and Doctor, which play the Observer role. The Patient class has a couple of variables, `_nurses` and `_doctor`, in which references to the nurses and doctor assigned to the patient are stored. The Doctor and Nurse class each has a variable in which references to the patients that the doctor/nurse are assigned to are stored. In addition, Doctor stores information about the current *heart rate* of each patient that the doctor is assigned to. Similarly, the Nurse class stores information about the *temperature* of each patient that the nurse is assigned to. The Patient class provides methods that can be used to obtain information such as the temperature and heart rate of the patient. In addition, the class has a method, `changeCondition()`, that changes the state of the patient to a new (random but legal) state. Next, this method invokes the `Notify()` method of Patient, which in turn calls the `Update()` operation provided by the Nurse and Doctor classes on each nurse and doctor that it has a reference to. These operations use the information-returning methods of Patient to update their information about this patient.

The *(varMap)*s required in the various role maps in the subcontract for Observer as specialized in Hospital are easily defined. For example, for Patient as Subject, the `_observers` map simply returns the union of `_nurses` and `_doctor`. More interesting are the definitions of the auxiliary concepts. Two definitions are needed for *Consistent()* since two different classes play the Observer role. Only one definition is needed for *Modified()* since both its arguments are Subject role and only one class plays that role. We provided straightforward definitions for these concepts –for example, *Modified(s1, s2)* is true if either the *temperature* or the *heart rate* in `s1` is different from that in `s2`. Given all of this information in the subcontract, MonGen produced the corresponding subaspect defining the various items that were abstract in the aspect generated from the pattern contract. When the resulting aspect and subaspect were woven with the code for the Hospital system and executed, the system ran as expected. Next we injected some faults into the implementation (such as `Patient.Notify()` not invoking `Update()` on one of the nurses observing it) and the system discovered them.

## 5. ADDITIONAL EVALUATION STUDIES

Evaluating the applicability and utility of our approach is an important focus of ongoing work. We have already applied the ideas presented here to several different patterns, and have begun using the resulting contracts, subcontracts, and monitors in our own work<sup>5</sup>. Here, we briefly summarize our experiences with two additional patterns, and consider scenarios in which the corresponding artifacts might be used.

We first consider Memento [6]. The intent of this pattern is to allow an originator to externalize its state in the form of a memento, so that it might later be used to restore the originator's state. One of the interesting concepts used in our contract for this pattern is *SameCopy*, a relation over two states of a memento. The relation captures whether these states record the same *essential* information about the originator. This concept is used in specifying the others requirements in the Memento role contract. As a result, our specification allows changes to a memento, so long as those changes preserve the *essential* information it records. Again, this is a case where our approach identifies dimensions of flexibility beyond what is allowed by the informal description — which suggests that mementos may never be changed.

Consider using this contract in a visualization project. To allow users to move backward in an animation, each visualization object would use the pattern to externalize its position in each frame. During the animation, a user might choose to include additional objects. When this occurs, existing memento objects would be modified by an object layout algorithm. The intent would be to allow users to rewind with the new object included, while keeping the original objects as close to their original positions (before the new object was included) as possible. After developing the subcontract for this system, we would use MonGen to generate the appropriate monitoring code. During system testing, we might use this code to detect violations of the pattern invariant — an indication that some memento has lost its *essential* information about the originator.

But how could this be if the aspects didn't detect any other contract violations? In a similar scenario in one of our own projects, further analysis revealed that the invari-

<sup>5</sup>Some of these artifacts are available for download [22].

ant was violated because our definition of *SameCopy* violated a constraint that requires definitions to be *transitive*. In the current example, several new objects may have been added during a single animation. While no single change caused a significant modification of any memento’s state, the *series* of changes did. The transitivity constraint on *SameCopy* guards against such changes that erode the information stored within a memento. This scenario reaffirms the importance of concept constraints, and shows that there is merit in monitoring violations of the pattern invariant, even though the invariant is guaranteed when the contract requirements are met.

The second case study focuses on Chain of Responsibility [6]. The intent of this pattern is to give multiple handler objects a chance to service the same request. The handlers are arranged in a chain, and each handler is passed the request in turn. In the contract for this pattern, we introduce a state variable in the Request role contract that allows us to record the handlers that have already received a request. We use these variables to impose a requirement that a request never pass through the same handler twice. In effect, we are able to recast the structural requirement of acyclicity as a behavioral requirement.

Note that this contract makes use of two constructs that were not discussed in Section 2 due to space limitations. First, it uses *auxiliary role variables*. These variables are similar to the types of role variables that we have considered, but represent *specification state*, rather than implementation requirements. As a consequence, these variables are provided by the contract itself, and are not mapped to player state in a subcontract. Second, it uses *specification actions* that update the values of auxiliary role variables. These actions are included as part of the appropriate ensures clauses in the role contracts.

We used PCL contract for this pattern in an extended version of the case study presented in Section 4. The pattern was used to arrange the nurse objects in a chain, and to handle requests concerning the patients that they observe. Again, we used MonGen to generate the monitoring code for this system based on the corresponding subcontract. We were able to use this code to detect cycles in the handler chain. While the defects were introduced artificially, the case study is interesting as it illustrates the utility of our monitoring approach in detecting *structural* errors.

## 6. RELATED WORK

Reenskaug’s OO modelling work [15] has had a significant influence on the design of our specification language. He describes a *role* as a projection of a class that focuses attention only on those aspects relevant to a particular area of concern. In our work, we use the notion of a role to model each type of object that participates in a pattern. As a result, our specifications focus only on those aspects of the object that are relevant to the pattern in question. Although the work presented in [15] considers the use of roles in modelling design patterns, it does not consider precise pattern characterizations or runtime monitoring.

Several other authors have considered pattern formalization. Eden *et al.* [4, 5] propose a higher-order logic formalism and graphical notation for expressing patterns as formulae. Each formula consists of a declaration of the participating classes, methods, and inheritance hierarchies, as well as a conjunctive statement of the relations among them. While

rich structural properties may be expressed, there is only limited support for behavioral properties. The formalism does not, for example, provide constructs for referring to pre- and post-conditional values, nor does it provide a concept analogous to our method call sequences.

By contrast, Mikkonen’s modelling approach [13] focuses exclusively on behavioral properties. In his approach, data classes model pattern participants, and guarded actions in an action system model their interactions. Relations are defined over the participants — similar to our use of auxiliary concepts — and the system actions associate and disassociate participants based on these relations. One limitation of the approach is that the separation of actions and data is structurally inconsistent with the OO paradigm. Consequently, most structural properties cannot be expressed. Further, the resulting specifications cannot be specialized to the needs of particular systems.

Helm *et al.* [9] propose a *contract* formalism for specifying object compositions. While there is some similarity between our approaches, there are also important differences. For example, although the formalism provides a construct analogous to our auxiliary concepts, it does not provide a way to impose conditions on the definitions that might be supplied. Further, although the formalism provides support for specifying the relative order in which method invocations must occur, the notion of a call sequence does not appear as a first class construct. It is impossible, for example, to quantify over a method call sequence to require that a particular method be invoked exactly once, or alternatively, that a particular method not be invoked at all. Finally, the formalism does not provide a construct analogous to our *others* clause. Hence, methods that are not explicitly required by a pattern could violate its intent.

Several authors have considered the question of how to incorporate runtime assertion monitoring in OO systems. The various approaches reported in [12, 18, 2], for example, describe language extensions and compiler tools for annotating programs with assertions, and generating the corresponding monitoring code. Other researchers have considered aspect-based approaches. In [11], for example, Lippert and Lopes report on their use of AspectJ in refactoring pre- and post-conditional assertion checking code. In [7], Gibbs and Malloy describe an approach to using aspects to monitor class invariants involving temporal properties in C++ code. To our knowledge, however, we are the first to investigate contract monitors for design patterns, as well as the automated generation of such monitors.

In [8], Hannemann and Kiczales describe an approach to *implementing* design patterns using AspectJ. In their approach, the object interactions required by the pattern are implemented as aspects, rather than being integrated into the code of the relevant classes. Interestingly, our monitoring approach also applies to these aspect-based implementations. Indeed, we were able to weave the monitoring code produced by MonGen with the aspect-based implementation of the Observer pattern presented in [8]. When we ran the resulting system, we discovered a subtle error in the implementation. The aspect code failed to invoke `Update()` when a new observer attached to a subject.

## 7. DISCUSSION

We have presented two contributions focused on automating the detection of pattern implementation errors. We first

described *PCL*, a pattern specification language designed to support runtime specification checking. *PCL contracts* allow us to capture implementation requirements and behavioral guarantees associated with a range of patterns. *PCL subcontracts* allow us to capture the ways in which patterns are specialized for use in particular systems. In this way, we are able to capture properties common across all applications of a given pattern, while accommodating the variation that occurs across those applications. Our second contribution was the design and implementation of *MonGen*, a tool for automating the generation of runtime monitoring code. Given the pattern contracts and subcontracts used in designing a particular system, *MonGen* produces a set of aspects that can be used to monitor the system's runtime behavior to determine whether the appropriate pattern requirements have been satisfied. We demonstrated our specification and monitoring approach in the context of a simple case study involving the Observer pattern.

An important aspect of our work is its applicability *across* the system lifecycle. The contracts and subcontracts used in designing a given system are valuable design artifacts that support program comprehension. Members of a maintenance team, for example, can study these artifacts to gain a more complete understanding of a system's design. As a result, these designers are less likely to violate the system's *design integrity*. Even so, given that errors might inadvertently be introduced, the subcontracts corresponding to the modified system can be used (in combination with the pattern contracts) to generate monitoring code appropriate to the new system. In fact, when the modifications do not affect the ways in which patterns have been specialized, the aspects produced by *MonGen* based on the *original* subcontracts may be used without change. Consider, for example, modifying the system in our case study to include a new Patient method that simulates the effects of a particular medication by reducing the patient's heart rate whenever it is invoked. By examining the role maps in the PND subcontract, the designer can quickly see that instances of the Patient class serve as subjects in instances of the Observer pattern. The designer must then determine the requirements that must be satisfied by a subject object's non-role methods. By examining the others clause in the Subject role specification, the designer can quickly see that since the new method modifies the state of subject — according to the definition of Modified supplied in the subcontract — it must also invoke the subject's Notify() method. If, however, this call is omitted by mistake, the aspects discussed in Section 4 can be reused without change to detect the error. This is because the modification does not affect the manner in which the Observer pattern was specialized.

Our work has focused on traditional design patterns for sequential object systems. The approach seems applicable, however, to a wider range of patterns. Patterns for distributed and networked systems are especially interesting. Consider, for example, the *Active Object* pattern presented in [19]. The intent of the pattern is to improve concurrency while simplifying synchronization by transforming system objects into threaded servers (or *actors* [1]). The key properties that must be specified involve the order in which messages must be exchanged between the threaded objects. The call sequence constructs provided by *PCL* are well-suited to expressing such conditions. Additionally, since the state conditions that trigger message transmission between the

objects vary from application to application, these conditions are suitably captured using the auxiliary concepts supported by *PCL*. These concepts may have to be generalized to accommodate buffered invocations that have not yet been handled. Further, the assertions expressed using these concepts may involve temporal operators. Supporting these extensions in *PCL*, and incorporating them into our monitor generation strategy, is part of our future work.

## 8. REFERENCES

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MITP, 1986.
- [2] L. Burdy *et al.* An overview of JML tools and applications. *STTT*, 2005. (to appear).
- [3] F. Buschmann *et al.* *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [4] A. Eden. Formal specification of object-oriented design. In *Proc., CSME-MDE*, 2001.
- [5] A. Eden. A visual formalism for object-oriented architecture. In *Proc., IDPT*, 2002.
- [6] E. Gamma *et al.* *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.
- [7] T. Gibbs and B. Malloy. Weaving aspects into C++ applications for validation of temporal invariants. In *Proc., CSMR*, pages 249–258, 2003.
- [8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc., OOPSLA*, pages 161–173, 2002.
- [9] R. Helm *et al.* Contracts: Specifying behavioral compositions in object-oriented systems. In *Proc., OOPSLA-ECOOP*, pages 169–180, 1990.
- [10] G. Kiczales *et al.* An overview of AspectJ. In *Proc., ECOOP*, pages 327–353, 2001.
- [11] M. Lippert and C. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc., ICSE*, pages 418–427, 2000.
- [12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- [13] T. Mikkonen. Formalizing design patterns. In *Proc., ICSE*, pages 115–124, 1998.
- [14] T. Parr and R. Quong. ANTLR: a predicated LL(k) parser generator. *Softw., Prac. and Exp.*, 25, 1995.
- [15] T. Reenskaug. *Working with Objects*. Prentice, 1996.
- [16] D. Riehle. Composite design patterns. In *Proc., OOPSLA*, pages 218–228, 1997.
- [17] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *TAPOS*, 2(1):3–13, 1996.
- [18] D. Rosenblum. A practical approach to programming with assertions. *IEEE TSE*, 21(1):19–31, 1995.
- [19] D. Schmidt *et al.* *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 1996.
- [20] N. Soundarajan and J. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Proc., ICSE*, pages 666–675, 2004.
- [21] N. Soundarajan *et al.* Specifying and monitoring design pattern contracts. In *Proc. SAVCBS (ICSE) Workshop*, pages 87–94, 2004.
- [22] B. Tyler *et al.* *MonGen: Monitor generator for pattern contracts*. [www.cse.ohio-state.edu/~tyler/MonGen](http://www.cse.ohio-state.edu/~tyler/MonGen).