Specifying and Reasoning About Design Patterns

Neelam Soundarajan
Computer Sc. & Eng. Dept.
Ohio State University
Columbus, OH 43210, USA

Jason O. Hallstrom
Computer Science Dept.
Clemson University
Clemson, SC 29634, USA

Tech Report: OSU-CISRC-8/05-TR54

Abstract

Design patterns are valuable both for designing and for documenting software systems. Patterns are usually described informally, in the style introduced by [9]. While such informal descriptions are certainly useful, in order to ensure that designers use the patterns correctly, and to reliably predict the behaviors that systems built using specific patterns will exhibit, we also need ways to reason precisely about patterns, and about the systems built using them.

This paper makes two contributions. First, we develop an approach to specifying patterns precisely. Second, we develop an approach which can be used by a designer to establish whether his or her system meets the requirements contained in the specifications of the patterns used in its design; and if it does meet the requirements, to arrive at conclusions about the behavior of the system. We illustrate the approach in the context of a detailed case study built using the *Observer* pattern, and an extended version of the system that also uses the *Chain of Responsibility* pattern.

A key aspect of many design patterns, the aspect that makes them applicable to a variety of systems, is their *flexibility*. Our approach, while allowing us to specify patterns precisely, also enables us to retain this flexibility. Indeed, the approach often enables us to identify additional dimensions of flexibility that are missing in the standard informal descriptions of patterns.

1 Introduction

Design patterns [2, 3, 30, 9, 19, 27] have, over the last decade, fundamentally altered the way we think about the design of large software systems. The use of patterns not only helps system designers exploit the collective wisdom and experience of the community as captured in the patterns, it also enables others studying the system in question to gain a deeper understanding of how the system is structured and why it behaves in particular ways. As Buschmann *et al.* [3] put it, "patterns support the construction of software with *defined properties* [emphasis added]". But if we are to fully realize these benefits, we must have ways to specify patterns formally so that the requirements that must be met when using a particular pattern, as well as the behavior that the system built using the pattern is guaranteed to exhibit, are precisely defined. We must also have techniques to show that a given system does meet the requirements defined in the specifications of the patterns underlying its design, so that we can rely upon the behaviors that the correct use of the patterns guarantees. Our goal in this paper is to develop an approach to reasoning about patterns and systems built using them that serves these purposes.

Let us first consider how patterns may be specified precisely. Typically, a pattern can be thought of as consisting of a number of *roles*, with one or more objects playing each role. For example, the Observer pattern, which we will consider in detail, has two roles, Subject and Observer. When a group of objects is interacting according to this pattern, one object will be enrolled to play the Subject role, and others will be enrolled as Observers. Using the pattern as intended requires the enrolled objects to interact with each other in ways that satisfy certain conditions in terms of the order in which various methods are called on particular objects, the conditions under which the calls are made, the effects the calls should have on the objects, etc. In order to express such requirements, a pattern specification or *pattern contract* in our approach consists of a role-level portion corresponding to each role of the pattern, and a pattern-level portion. The role-level portion specifies conditions that the classes playing the particular roles of the pattern must satisfy with respect to the behaviors that relevant methods must exhibit, potentially including conditions on the method calls that they must in turn make during their execution. The pattern-level portion includes an *invariant*, an assertion over the various role states that represents the behavior that a system built using the pattern is guaranteed to exhibit.

The standard informal description [9] of the Observer pattern makes it clear that the Subject must include a Notify() method that invokes the Update() method on each of the objects enrolled as observers[1] of the sub-

[1]We use names starting with uppercase letters such as Subject for roles; and the corresponding lowercase names such as subject to refer to the individual objects that play these roles. We also use names starting with uppercase letters for patterns and classes. Occasionally, we also use the name of a pattern for one of its roles, as in the case of

ject. When is the Notify() method itself called? This question is addressed in the commentary in the informal description of the pattern: "... subject notifies its observers whenever a change occurs that could make its observers' states inconsistent with its own." What is not clear is how the subject will know when its state has become inconsistent with that of an observer. Indeed, what does it mean to say that the subject state is *inconsistent* with that of an observer? The informal description of Observer also suggests that if the pattern is applied as intended, the view of each observer will remain consistent with the subject's state – but again, *consistent* in what sense? Our contract for the pattern will provide precise answers to these questions. In the absence of such precision, there is a danger that different members of a design team working on the same system may have different, mutually incompatible notions of what it means for the subject state to be consistent with that of a given observer's state. This, in turn, may well lead to the system exhibiting unexpected behaviors.

But there is an inherent risk in formalizing design patterns in that *flexibility*, which is the hallmark of many design patterns, may be lost [26]. For example, in the case of Observer, if we adopt one definition for the notion of *consistency*, the pattern may not be usable in systems that have a different notion of this concept. As we will see, our approach, while allowing us to provide a precise characterization of the pattern, also enables us to retain the flexibility contained in the pattern. One key to preserving this flexibility is based on the notion of an *auxiliary concept*, which is a central element of our approach. An auxiliary concept is essentially a relation on the states of one or more objects playing roles in a given pattern. Thus, *Consistent*(s1, o1), a relation on the state s1 of the subject and the state o1 of an observer, will be one of the auxiliary concepts declared in our contract for Observer. Flexibility is achieved by allowing the definitions of the auxiliary concepts to be tailored to the needs of the particular systems built using the pattern. Precision is achieved by expressing the contract in terms of the auxiliary concepts, and by restricting the application-specific definitions that may be supplied. These restrictions are imposed using *concept constraints*. We will see the importance of these constraints when we consider particular pattern contracts.

We will see in the next section that in addition to auxiliary concepts, there are a number of other aspects of our approach that contribute to retaining the flexibility of design patterns. Indeed, the very process of formalizing a pattern in our approach occasionally enables us to identify *additional* dimensions of flexibility that may be missing from the standard informal descriptions. We will see several examples of this in the paper.

Let us now turn to the question of how a designer may reason about a given system built using a particular pattern to show that it satisfies the requirements specified in the pattern contract, and how he or she may draw appropriate conclusions about the behavior that the system may be expected to exhibit as a result of using the pattern correctly. We will use the term *pattern specialization* to refer to the manner in which a given pattern is adapted for use in a particular system. We will consider a simple system, SimSickCity, with three classes, Patient, Nurse, and Doctor. This system will use the Observer pattern. Instances of Patient will play the Subject role, and instances of Nurse and Doctor will play the Observer role. In reasoning about this system, the designer will be required to define a *pattern subcontract* that captures the precise manner in which the pattern is specialized for use in this system. For this, the subcontract will specify the exact manner in which the classes, Patient, Nurse, and Doctor, map to their respective roles. It will also provide definitions, appropriate to this specialization of the Observer pattern, for the auxiliary concepts used in the pattern contract. The designer will then have to demonstrate, given the mappings in the subcontract, that the requirements specified in the pattern contract are satisfied. This will involve showing that the constraints imposed on the auxiliary concepts, the requirements imposed on the methods of the classes playing the individual roles, and other requirements whose details we will see in the next section, are satisfied. The designer may then appeal to the invariant and other properties guaranteed by the correct use of the pattern, again specialized on the basis of the mappings and concept definitions in the subcontract, to arrive at conclusions about the behavior that the system can be expected to exhibit.

Thus, the pattern contract specifies the aspects of the pattern that are common across every use of the pattern, and a subcontract specifies a particular specialization. Note also that at any given time during a system's execution, there may be several sets of objects interacting according to the particular pattern contract, specialized according to the particular subcontract. Thus in the SimSickCity system, at any given time, we may have several groups of objects, each group containing a patient object and a number of nurse/doctor objects *observing* the patient. We will call such a group of objects an *instance* of this pattern specialization. Note further that a system may also use several specializations of a given pattern. In such a case, each group of objects interacting according to a particular specialization (i.e., each instance of that specialization) will satisfy the pattern contract specialized according to the mappings provided in

the Observer role of the Observer pattern. In such cases, the context will clarify which is intended.

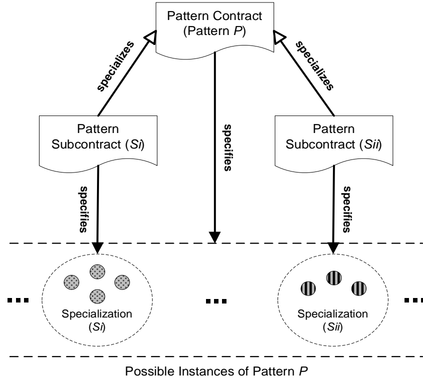

Fig. 1. Patterns, Specializations, Instances

the subcontract for that specialization. This is illustrated in Fig. 1. The figure represents a pattern P and two specializations of P, *Si* and *Sii*. The subcontracts for *Si* and *Sii* are represented as *specializing* the contract for P. In the lower part of the figure, between the two dashed lines, we have a representation of the runtime picture. We have two ovals corresponding respectively to *Si* and *Sii*. Inside each oval, we have small circles representing the individual *instances* of the *Si* and *Sii* specializations. Each of these small circles essentially represents the various objects interacting according to P as specialized by *Si*/*Sii*. The requirements listed in P's contract apply to each of these instances. P's contract, as specialized by the subcontract for *Si*/*Sii* applies to the instances in the oval corresponding to *Si*/*Sii*, respectively.

Moreover, practical systems are often built using several different patterns. The figure could be extended to show this by depicting another pattern Q, its specializations, with instances of those specializations added to the lower part of the figure inside ovals corresponding to Q's specializations. The reasoning approach we develop in the rest of the paper is designed to handle such situations. To illustrate this, we will extend the functionality required of the SimSickCity system as follows. We will require that there be a particular nurse object to which inquiries concerning the current state of *any* patient object may be sent. If this nurse object is able to provide the requested information –which will be the case if this nurse is currently an observer of this patient– it will return the information; otherwise, it will forward the request to another nurse object in the system; this nurse object, in turn, will return the requested information if it is able to do so, i.e., is currently an observer of the patient, else will forward it to another nurse; and so on. This implementation strategy corresponds to an application of the *Chain of Responsibility (CoR)* pattern [9] with the initial nurse object at the head of the chain. We will see how our approach allows us to reason about this extended system.

Patterns are usually classified into three groups, *structural*, *creational*, and *behavioral* [9]. But often, patterns classified as behavioral have structural aspects to them, and, conversely, patterns classified as structural have important behavioral aspects associated with them, etc. Buschmann et al. [3] also note this; in describing how a pattern provides a way to solve a recurring problem, they note "...such a solution includes two aspects. Firstly, ... a certain structure, a spatial configuration of elements. ... Secondly, every pattern specifices runtime behavior." Thus, for example, the Observer pattern requires the Subject role to maintain a set of references to the objects playing the Observer role, and this state component is essential in implementing the behavior that the correct use of Observer ensures. But, at the same time, there is a structural element to this. Similarly, the Composite pattern, while it is classified as a structural pattern, ensures, if used correctly, important behavioral characteristics in the systems built using the pattern. Indeed, this is usually a key consideration in using any pattern. Thus while our focus is on specifying the behavioral requirements associated with using particular patterns, and the behavioral guarantees that the correct use of those patterns ensure, our work is important for all types of patterns, whether they are classified as behavioral, structural, or creational. We should also stress that our intent is to complement, not replace, informal descriptions and UML diagrams that are currently used in describing patterns.

During the past few years, several authors have considered the question of formalizing design patterns. We present a detailed analysis of related work in Section 5. At this point, however, it is worth mentioning Reenskaug's work on *role models* [25]. Reenskaug describes a role model as a projection of a class with respect to a particular area of concern. This projection allows us to focus only on those aspects of an object collaboration that are relevant to an area of concern under consideration. For example, an area of concern for a system could be the question of how a group of observer objects keep track of the changes in a subject. An important point is that a given object playing the role of an Observer may have other aspects to it that are not relevant to this area of concern, and will therefore be excluded from the corresponding role description. Although Reenskaug is not concerned with formal specifications, his concept of a role model had an important influence on our approach to formalizing patterns. In a sense, some of the information in a pattern contract in our approach can be considered as a formalization of a role model.

In an earlier paper [32], we considered the formalization of design patterns, and provided a preliminary

foundation for the more complete results presented here. There are a number of important differences, however. First, although [32] focused on design pattern specifications, we did not consider the development of a formal specification language. Second, in [32], we did not consider the specialization of design patterns, and by consequence, did not consider the notion of a pattern subcontract. Third, [32] only briefly considers the question of how to reason about a system using a set of contracts, and does so under the requirement that the system code is appropriately instrumented using a specialized pattern meta-language. Finally, the concepts discussed in [32] do not support multiple instances of the same pattern. Even so, although the work presented here is a significant extension of the ideas presented in [32], it builds upon some important principles outlined in that paper.

The rest of the paper is organized as follows. In the next section, we develop the key specification language constructs, and describe the details of pattern contracts and subcontracts. We also introduce notations that are useful in the expression of pattern contracts. In Section 3, we present a case study in which we consider a simple system built using the Observer pattern, and an extension of that system built using the Chain of Responsibility pattern. We develop the complete pattern contract and subcontract for Observer, and discuss the key portions of the contract and subcontract for the Chain of Responsibility. We show how to use these contracts and subcontracts in determining whether the patterns have been applied correctly, and in reasoning about the resulting system behaviors. In Section 4, we briefly discuss our experiences applying the specification and reasoning approach to other design patterns. Section 5 presents a discussion and analysis of related work. In the final section, we summarize our results, and discuss the incorporation of our specification and reasoning approach in a *pattern-centric* software process. We also discuss initial results in using runtime monitoring to detect pattern contract violations. We conclude with pointers to future work.

2 Pattern Contracts and Subcontracts

We begin by reviewing some of the terminology introduced in the preceding section, and introduce some additional terms that will be important in describing our formalism. Consider a system S built using a pattern P. Let SP be the *specialization* of P used in S. At any particular time during the execution of S, we may have several distinct groups of objects, with the objects in each group interacting according to P (as specialized in SP). Each such group is an *instance* of SP. Where there is no possibility of confusion, for example if there is only one specialization under discussion, we will also refer to the group as an instance of the pattern.

Each object in a specialization instance is enrolled to play a particular role of the corresponding pattern. Thus, in the SimSickCity system, any Nurse objects in a given specialization instance will be enrolled to play the Observer role, whereas the Patient object will be enrolled in the Subject role. During execution, additional objects may join a given instance. For example, a new nurse may be assigned to watch a particular patient. In this case, we say that the object *enrolls* in the given instance. Note that a given object may simultaneously be enrolled in several instances of the same or several distinct patterns. Thus, a nurse may be assigned to watch several patients, and would therefore be enrolled in each of the corresponding instances. In the extended version of the system that we will consider, the nurse will also be enrolled in an instance of the Chain of Responsibility pattern. Note, however, that a given object cannot be enrolled more than once in a given specialization instance, either in the same role or in distinct roles. An object may also *disenroll* from a pattern instance. Thus in the SimSickCity system, a nurse assigned to watch a given patient may be unassigned by invoking the removeNurse() operation on the patient, and will then no longer be enrolled in that instance. Note, however, that the nurse in question will continue to be enrolled in other instances of the pattern.

The contract for the pattern P and the subcontract corresponding to SP will together specify how new instances of the specialization are created. The contract will specify a particular role method or constructor that must be called, and the conditions that must be satisfied at that point, for a new instantiation to occur. The subcontract will provide the mapping that will identify the actual method (of the class playing the role) that maps to the specified role method. The pattern contract will also specify, for each role, the role method or constructor that must be called, and the conditions that must be satisfied at that point, for an object to enroll to play the role in an existing specialization instance. The subcontract will specify the actual class method that corresponds to the specified role method.

When specifying a pattern, we will identify one of the constituent roles as the *lead role*. The instantiation action specified by the contract will be a method or constructor call to an object that will play this role upon instantiation. When this action is performed, and the appropriate conditions are satisfied, a new pattern instance will be created with only this particular object enrolled in that instance. We call this object the *lead object*. The lead object serves as a convenient handle to refer to the corresponding specialization instance. Thus, for exam-

ple, when specifying how an object enrolls to play a particular role in an existing pattern instance, we identify the instance in question by specifying the corresponding lead object. To prevent ambiguous instance handles, we require that the same object not be the lead object of more than one instance of a given pattern.[2]

We should also note that for particular patterns, additional objects may be allowed to enroll in the lead role. Whether or not this is possible depends on the corresponding enrollment condition. If, for example, the role enrollment condition is specified as false, or equivalently, the enrollment clause is altogether omitted, an object may enroll in the lead role only at the point of pattern instantiation, and additional objects may not enroll. In the Observer contract, for example, the Subject role will be specified as the lead role, and the Subject enrollment clause will be omitted, indicating that exactly one subject object will be enrolled in any given instance of the pattern, and that object may be used to identify the instance. Note that even if additional objects are allowed to enroll in the lead role, only the object that enrolled at the point of pattern instantiation serves as the lead object.

The rest of this section is organized as follows. In Section 2.1, we introduce our formalism for specifying pattern contracts. In Section 2.2, we consider subcontracts corresponding to pattern specializations. In Section 2.3, we introduce some notations that simplify the expression of pattern contracts and subcontracts.

2.1 Pattern Contracts

The partial grammar for specifying pattern contracts appears in Fig. 2. The contract for a pattern specifies (*i*) the name of the pattern (⟨*pId*⟩[3]), (*ii*) an auxiliary concept block, (*iii*) a contract for each role, (*iv*) an *instantiation clause* that specifies how a new instance of the pattern is created, and (*v*) an *invariant* for the pattern.

The ⟨*auxConceptBlock*⟩ declares the auxiliary concepts used throughout the pattern contract. Note that we have omitted some simple productions. Thus, ⟨*auxConcepts*⟩ represents any number of repetitions of the ⟨*auxConcept*⟩ production; ⟨*roleContracts*⟩ represents repetitions of the ⟨*roleContract*⟩ production; etc. For each *auxiliary concept*, we specify its name (⟨*auxId*⟩) and the list of role names over which the concept is defined. The same role name may appear multiple times in this list. For example, in the contract for the Observer pattern, we will use the *Modified*() auxiliary

```
⟨patternContract⟩     ::= pattern ⟨pId⟩ contract {
                              ⟨auxConceptBlock⟩
                              ⟨roleContracts⟩
                              ⟨instantiation⟩
                              ⟨invariant⟩ }
⟨auxConceptBlock⟩     ::= auxiliary concepts:
                              ⟨auxConcepts⟩
                              ⟨auxConstraints⟩
⟨auxConcept⟩          ::= ⟨auxId⟩ (⟨rIds⟩);
⟨auxConstraints⟩      ::= constraints:
                          ... predicate on aux. conc. ...
⟨roleContract⟩        ::= [lead] role ⟨rId⟩ contract {
                              ⟨roleStateSpec⟩
                              ⟨namedMethodSpecs⟩
                              ⟨othersSpec⟩
                              ⟨enrollment⟩
                              ⟨disenrollment⟩ }
⟨roleStateSpec⟩       ::= ... role fields ...
⟨namedMethodSpec⟩     ::= ... standard method specs.
                              with requires, preserves,
                              and ensures clauses ...
⟨othersSpec⟩          ::= others:
                              ... method spec. with
                                  only preserves and
                                  ensures clauses ...
                                  ... see discussion ...
⟨enrollment⟩          ::= ... see discussion ...
⟨disenrollment⟩       ::= ... see discussion ...
⟨instantiation⟩       ::= ... see discussion ...
⟨invariant⟩           ::= invariant:
                              ... assertion on roles and
                          concepts...
```

Fig. 2. Grammar of Pattern Contracts

concept, which represents the notion of whether a particular state of the subject contains the same essential information as some other state of the subject. Hence, this concept will have Subject listed twice among its argument role names. On the other hand, *Consistent*(), the other auxiliary concept used in the Observer contract, will involve the state of the subject and the state of an observer. This concept corresponds to whether the states of the two objects are consistent with one another. Each auxiliary concept is a boolean function over its arguments. The various auxiliary concepts will be used throughout the specification, including (*i*) in specifying the method contracts for each role, (*ii*) in specifying the enrollment and disenrollment clauses, (*iii*) in specifying the instantiation clause for the pattern, and finally, (*iv*) in specifying the pattern invariant.

Next we have the *constraints* that must be satisfied

[2] It would be possible to extend the formalism to remove this requirement but it is satisfifed in the case of all the standard design patterns we have considered.

[3] We use ⟨*pId*⟩ to stress that this is the name of a pattern; similarly, ⟨*rId*⟩ will be the name of a role; etc.

by any definitions supplied for the auxiliary concepts. As mentioned in the introduction, and as we will see in the Observer contract in the next section, if the concept definitions corresponding to a particular specialization do not satisfy certain constraints, the intent of the pattern may be violated even if the system satisfied all of the other requirements specified in the contract. The relations that must be satisfied apply to *all* specializations of the pattern, and therefore belong in the pattern contract, rather than in each of the subcontracts. We refer to these relations as *concept constraints*, predicates over the relevant auxiliary concepts.

Next let us consider the details of the ⟨*roleContract*⟩ production. Each role contract specifies the name of the role (⟨*rId*⟩), with exactly one role being labelled as the lead role of the pattern. The first part of the role contract specifies the *role state*, which consists of a list of typed field declarations. The syntax is the same as for standard field declarations, and we omit the details. We should note that the fields listed in the role state need not be explicitly present in the class that ultimately plays the role in a given specialization. Instead, as we will see, the subcontract for the specialization will include a map that specifies how the actual variables of participating classes map to the role-fields.

Following the specification of the role state, we have the specifications of the "named" methods. The *named* methods are the ones that an object playing this role must include in order to support its part in the pattern. For example, the Observer role of the Observer pattern must provide an Update() method; the Subject role must include an Attach() method; etc. Suppose, as specified in a subcontract of a given pattern P, the class C plays the role R. C will then be required to provide each of the named methods listed in the role contract of R. Or more accurately, as in the case of the role-fields, the subcontract must map each of the named methods in R to the appropriate methods of the class whose instances will play the role. These methods will be required to satisfy the corresponding specifications listed in the role contract. Each method specification consists of (*i*) a *requires* clause corresponding to the method's pre-condition, (*ii*) a *preserves* clause that identifies those state components and method arguments that must be unaffected by an invocation of the method, and (*iii*) an *ensures* clause corresponding to the method's post-condition.

C will often provide additional methods beyond those used to support the pattern. These "non-role" methods are C's "other" methods. If these *other* methods were not suitably designed, the intent of the pattern could be compromised. For example, if the class playing the Subject role in a specialization of Observer were to include a method that changed or destroyed the information about the set of attached observers, the system would clearly violate the intent of the pattern. This would be true even if all of the methods explicitly listed in the Subject role contract were implemented correctly. The others specification imposes conditions to prevent such problems, and these conditions must be met by all non-role methods. Note, however, that these conditions involve only preserves and ensures clauses since these clauses capture the changes the methods effect. No conditions are imposed on their requires clauses.

Next we have the ⟨*enrollment*⟩ clause. In the case of the Observer pattern, an object enrolling as an observer in a particular pattern instance will do so by invoking the Attach() method on the subject object playing the lead role for that instance. In general, in specifying how an object enrolls in a pattern instance, we must specify the object to be enrolled, the method or constructor that must be called, the pattern instance into which the object is enrolling (by specifying the corresponding lead object), and the enrollment conditions that must be satisfied. The syntax for enrollment is:

```
⟨enrollment⟩ ::=
  enrollment: ⟨rId⟩.⟨mId⟩(⟨args⟩)
    lead: ( target | source | ⟨arg⟩ | ... code ... )
    enrollee: ( target | source | ⟨arg⟩ | ...code... )
    pre-cond: ... assertion on lead, enrollee, etc. ...
    post-cond: ... assertion on lead, enrollee, etc. ...
| enrollment: new ⟨rId⟩(⟨args⟩)
    lead: ( source | ⟨arg⟩ | ... code ... )
    enrollee: ( newOb | source | ⟨arg⟩ | ...code... )
    pre-cond: ...
    post-cond: ...
```

The first alternative in the production corresponds to the case when enrollment is associated with the invocation of a particular role method. ⟨*rID*⟩ denotes the name of the role, ⟨*mID*⟩ denotes the name of the role method, and ⟨*args*⟩ denotes the list of arguments passed to the method. The lead clause identifies the lead object of the pattern instance. There are several possibilities here. First, the object may be the target of the method call (*i.e.*, the object on which mId() is invoked). Second, the object may be the source of the method call (*i.e.*, the object whose method initiated the method call). Third, the object may be one of the arguments passed to the method call. Finally, the object may be some other object, perhaps one that the *source* object has a reference to. In this case, a segment of *code* will return the particular object that is to serve as the lead object. An alternative approach would have been to use a mathematical map that gives us the lead object. The enrollee clause, which specifies the enrolling object, is defined

analogously. Note, however, that the lead object must be distinct from the enrollee object. As we noted earlier, no object may play multiple roles in the same pattern instance.

The enrollment pre-condition is an assertion that must be satisfied at the point an object enrolls to play the role. Note, however, that the method in question will necessarily have a corresponding specification in the relevant role contract. This specification may include a non-trivial pre-condition. That pre-condition, however, as well as the rest of the pattern contract, only applies to objects that have *already* enrolled in a pattern instance. It is for this reason that we specify a pre- and post-condition as part of the enrollment clause. (The same is true of the instantiation clause.) An alternative would have been to use the role method specification included in the role contract in place of the enrollment pre- and post-conditions. However, having a separate set of conditions allows us to cater to systems in which different requirements need to be imposed when the method is used for enrollment as compared to when it is invoked by an object that is already participating in a pattern instance.

An object may also enroll at the time of its creation, or at the time of another object's creation. This possibility is captured by the second alternative within the production. We are again required to specify the lead object and the enrollee object. And again, there are several possibilities. The enrollee object may be the object just created (newOb), the source of the call to the constructor, one of the arguments to the constructor, or some other object specified by the appropriate *code* fragment. Similarly, the lead object may be the source, one of the arguments to the constructor, or another object specified by the appropriate *code* fragment. Note that the newly constructed object cannot be the lead object since the pattern instance, and hence the lead object, must already exist for an object to enroll in it.

The syntax for the disenrollment clause is similar, except that it cannot be associated with the creation of a new object.

⟨*disenrollment*⟩ ::=
  disenrollment: ⟨*rId*⟩.⟨*mId*⟩(⟨*args*⟩)
    lead: (target | source | ⟨*arg*⟩ | … *code* …)
    disenrollee: (source | target | ⟨*arg*⟩
                | … *code* …)

Note that unlike the enrollment case, the object in question is already enrolled in the pattern instance at the time of the call to ⟨*rId*⟩.⟨*mId*⟩(). Hence, it must satisfy the pre-condition specified for this method in the appropriate role-contract. We therefore do not need to specify a pre-condition as part of the disenrollment clause. Moreover, once the object disenrolls, it is no longer part of the pattern instance, and is therefore not required to abide by the pattern specification. Consequently, we do not need to specify a post-condition.

Note further that a role contract need not include a disenrollment clause. This is the case if an object, once enrolled to play a particular role, remains enrolled until either the object or the pattern instance is destroyed. The latter case occurs when the lead object for that instance is destroyed or disenrolls. Again, the enrollment clause may also be omitted, but only in the case of the *lead* role. As we noted earlier, this case applies when the only way for an object to enroll in a particular role is at the point of pattern instantiation. In that case, the ⟨*instantiation*⟩ clause, which we consider next, also serves as the ⟨*enrollment*⟩ clause.

⟨*instantiation*⟩ ::=
  instantiation: ⟨*rId*⟩.⟨*mId*⟩(⟨*args*⟩)
    lead: (target | source | ⟨*arg*⟩ | … *code* …)
    pre-cond: …
    post-cond: …
  | instantiation: new ⟨*rId*⟩(⟨*args*⟩)
    lead: (newOb | source | ⟨*arg*⟩ | … *code* …)
    pre-cond: …
    post-cond: …

The instantiation clause is similar in structure to the enrollment clause. The first production alternative captures the case where a new pattern instance is created by invoking a specified role method. Pattern instantiation can also be associated with the creation of an object, as in our case study where an instance of the pattern will be created each time a `patient` object is created. This possibility is captured by the second alternative of the production. In either case, we must specify the lead object that will be used to identify the new pattern instance. This object must of course be an instance of the role that has been marked as the lead role. The instantiation pre- and post-conditions are analogous to the enrollment pre- and post-conditions.

It may seem that by specifying the role methods associated with pattern instantiation and role enrollment, we run the risk of compromising pattern flexibility. The argument for this position would be that the formalism cannot handle systems in which the instantiation method, for example, has a different name or signature than the one listed in the pattern contract. However, this is certainly not the case. As we will see in Section 2.2, the *method maps* that appear as part of the subcontracts for individual specializations of the pattern will allow us to cater to such variations. We will see examples of this customization in our case study.

The final item in the pattern contract, following the

instantiation clause, is the *pattern invariant*, a predicate defined over the states of the various roles, and expressed in terms of the auxiliary concepts listed in the auxiliary concept block. In general, within the same pattern instance, multiple objects may be enrolled in the same role, and the invariant will have to refer to the states of all of these objects. The notations that we introduce in Section 2.3 will allow us to express such conditions conveniently. The invariant is guaranteed to be satisfied whenever control is not inside one of the methods acting on an enrolled object. This is of course true only if the various requirements listed in the role contracts, constraints, etc., are satisfied. In effect, the invariant is the formal version of the "defined properties" described in [3] that the correct use of the pattern ensures.

2.2 Pattern Subcontracts

The grammar for subcontracts corresponding to pattern specializations appears in Fig. 3. A subcontract specifies (*i*) the name of the pattern specialization (⟨*sId*⟩), (*ii*) the name of the pattern (⟨*pId*⟩) it specializes, (*iii*) a set of role maps, one corresponding to each role of the pattern, and (*iv*) a set of definitions for the auxiliary concepts declared in the corresponding contract.

⟨*subContract*⟩ ::= specialization ⟨*sId*⟩ : ⟨*pId*⟩
  subcontract {
  ⟨*roleMaps*⟩
  ⟨*auxConceptDefBlock*⟩ }
⟨*roleMap*⟩ ::= rolemap ⟨*cId*⟩ as ⟨*rId*⟩ {
  ⟨*leadObjRel*⟩
  ⟨*stateMap*⟩
  ⟨*interfaceMap*⟩ }
⟨*leadObjRel*⟩ ::= lead relation:
  ...relation on lead object and role object ...
⟨*stateMap*⟩ ::= state: ⟨*fieldMaps*⟩
⟨*fieldMap*⟩ ::= ⟨*rfId*⟩ = {... code ... }
⟨*interfaceMap*⟩ ::= methods: { ⟨*methodMaps*⟩ }
⟨*methodMap*⟩ ::= ⟨*rmId*⟩(⟨*rmArgs*⟩) :
  ⟨*cmId*⟩(⟨*cmArgs*⟩) {⟨*argMaps*⟩}
⟨*auxConceptDefBlock*⟩
  ::= auxiliary concepts: ⟨*auxConceptDefs*⟩
⟨*auxConceptDef*⟩ ::= ⟨*auxId*⟩ (⟨*cArgs*⟩) {... code ... }

Fig. 3. Grammar of Pattern Subcontracts

A *role map* is specified for each class (⟨*cId*⟩) whose instances may play the role (⟨*rId*⟩) in an instance of the specialization. Each role map specifies how objects that are instances of the corresponding class can, in effect, act as instances of the role when participating in the pattern. Note, however, that a given instance of the class (⟨*cId*⟩) may *simultaneously* play this role in multiple instances of this specialization. In our case study, for example, a given nurse object may be simultaneously observing multiple patients, thus playing the Observer role in the instances corresponding to each of those patients. Each role map must therefore specify how objects of type ⟨*cId*⟩ act as instances of the role ⟨*rId*⟩ in *each* of their respective specialization instances. Again, each specialization instance will be identified by its corresponding lead object. We use the keyword "lead" to refer to the current specialization instance under consideration (via the lead object used as a proxy for the instance). This is somewhat analogous to the keyword "this" used in OO languages when referring to an object instance from within its corresponding class implementation.

Let us first consider the *state map* portion of a role map. The state map consists of a set of *field maps*. Each field map corresponds to a role-field specified in the role contract for this role in the corresponding pattern contract (⟨*pId*⟩). A field map identifies the name of a particular role-field (⟨*rfId*⟩), and specifies a *code* fragment that takes the current state of the ⟨*cId*⟩ object and returns the value of this role field when the object is *viewed as* an instance of ⟨*rId*⟩ in a particular pattern instance. In simple cases, we will have a corresponding field in the class for each field of the role, and this code fragment will simply return the value of that field. In general, however, more complex relations may be expressed in our formalism. The reason we chose to express this relation as a piece of code, and the reason we chose to use code fragments elsewhere in the subcontract, is because of our desire to make the formalism more usable to practitioners who are likely to prefer expressing the required maps in the form of code rather than in the form of mathematical relations. This is especially important in a subcontract since a subcontract corresponds to the specialization of the pattern used in a particular system, and will have to be developed by the team responsible for the system. By contrast, the contract for any pattern is a one-time effort by the community; we will return to this point in the final section. We should note, however, that the code fragments corresponding to these elements in the subcontract must not, of course, result in any *side-effects* when they are executed. That is, the code fragments are intended to substitute for mathematical expressions, and must not modify any of the objects in the system.

When a participating object may be simultaneously enrolled in multiple specialization instances, it must be possible to differentiate those portions of the object's state relevant to each instance. To achieve this, the lead object corresponding to each specialization instance will

be used as an *index* into the state of the role object, to project out those portions of the object's state relevant to the corresponding instance. Consider, for example, an object playing the Observer role in multiple instances of the Observer pattern. When evaluating *consistency* in the context of a particular pattern instance – say, at the termination of the Observer.Update() method – we will require that the observer in question be consistent with its subject. Note, however, that this requirement only involves the portion of the observer's state that records information about *this* subject – not the portions concerning other subjects that this object may be observing. Therefore, when reasoning about the *consistency* of an observer in the context of a particular specialization instance, the subject serving as the lead object for that instance will be used to project out the relevant portion of the observer's state. We will see an example of this in our case study. There, a patient object will serve as the lead object in each specialization instance, and this object will be used to index into the state of a given nurse or doctor object to project out the state information relevant to a particular instance. To ensure the correctness of the indexing, we must be sure that the state of the observer in question contains information relevant to the patient in question. Similar conditions will be required in any system where a given object may be simultaneously enrolled in multiple instances of the same specialization. This type of condition is captured by the ⟨*leadObjRel*⟩ clause, which defines a suitable relation over the lead object and the role object under consideration. In our case study, this relation will simply state that the given patient must be included in the list of patients that the particular nurse or doctor object stores references to.

The *interface map* defines the mapping between the *named* methods of the role and the corresponding class methods. For each named role method (⟨*rmId*⟩), its *method map* specifies the corresponding class method (⟨*cmId*⟩) that serves as the role method in instances of the specialization. The method map also provides an *argument map* that specifies how the class method's arguments (⟨*cmArgs*⟩) map to the role method arguments (⟨*rmArgs*⟩). In some cases, the class method may provide fewer arguments than the role method to which it corresponds. In such a case, the designer may specify an argument mapping from the state of the class to the relevant role method argument. We will see an example of this in our case study. The production for ⟨*argMap*⟩ is similar to that for ⟨*fieldMap*⟩, and we omit it. And, as in the case of field maps, these mappings are typically straightforward, with each argument of the role method mapping to an argument of the class method. In some cases, however, a given role method may be mapped to more than one class method. In our case study, for example, the Attach() method of the Subject role will be mapped to both the addNurse() and the assignDoctor() methods of Patient. When we consider the corresponding subcontract in Section 3.3, we will see the slightly extended ⟨*methodMap*⟩ syntax used to accommodate these scenarios.

The final portion of a subcontract defines the appropriate auxiliary concept definitions. For each auxiliary concept (⟨*auxId*⟩), and each combination of classes that play the various roles that appear as its parameters, we must provide a corresponding definition. As in the case of the *field maps* and *method maps*, we provide suitable code fragments corresponding to each definition. The code fragments perform the appropriate comparisons across instances of the classes mapped to each role, and return *true* or *false* to indicate whether the corresponding relation is satisfied. Recall that these definitions must satisfy the *concept constraints* specified in the pattern contract.

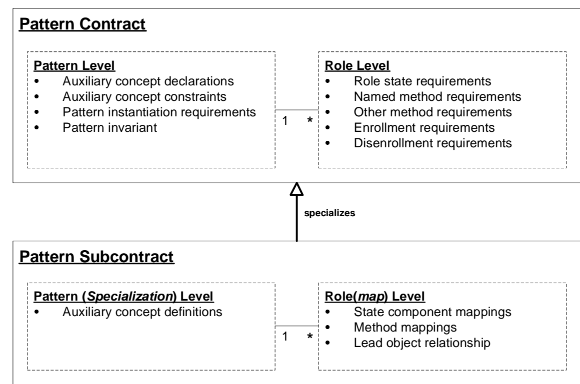

Fig. 4. Pattern Contracts and Subcontracts

Fig. 4 summarizes the information contained in pattern contracts and subcontracts. In essence, a pattern contract provides a *parameterized* formalization of a given pattern. The parameterization structure includes (*i*) a parameterized set of specifications for the individual role methods, (*ii*) parameterized enrollment and disenrollment clauses, (*iii*) a parameterized pattern instantiation clause, and, perhaps most important, (*iv*) a parameterized pattern invariant. The auxiliary concepts declared by the contract serve as the primary parameters, and the field and method maps serve as additional parameters. For any particular system that uses the pattern, the corresponding subcontract defines the actual parameters via the definitions of the auxiliary concepts, field maps, and method maps. Thus, while the pattern contract specifies the pattern precisely, the parameterization structure ensures that flexibility is not compromised.

2.3 Role Players and Call Sequences

When writing pattern contracts and subcontracts, we often need to refer to the objects playing the various roles in a given pattern instance. For this purpose, we introduce a special symbol, players, which denotes a *vector* of all the objects enrolled in various roles, in the order that they enrolled. players will contain an ordered pair corresponding to each enrollee, consisting of a reference to the object playing the role, as well as the name of the particular role. If an object xx disenrolls, xx will be removed from players, and any objects that enrolled after xx, and occupying locations in players past the one occupied by xx, will be "moved down" one position.

Individual elements of players can be referred to using standard indexing notation. Thus, players$[0]$ denotes a pair consisting of the first (*i.e.* lead) object to enroll and its associated role. players$[i{:}j]$ denotes the *slice* consisting of the ith through the jth elements. If j is missing, the slice consists of all the elements from the ith through the last element. If i is missing, the slice is from the 0th element through the jth. We introduce some additional functions on players that will be useful in writing contracts and subcontracts:

players.objv: Denotes a vector of enrolled objects, omitting their role names. When the ordering information is not important, we denote the *set* of objects enrolled as players.objs.

players.rolev: Denotes a vector of filled role names, omitting information about the objects enrolled in those roles. Again, when the ordering information is not important, we denote the set of roles as players.roles. More precisely, since many objects may be enrolled to play a given role, players.roles is not a set, but a *multi-set*. Note, by contrast, that since an object cannot play two roles, nor enroll in the same role twice in any pattern instance, players.objs is necessarily a set.

players.*Rv* (where R is a particular role name): Denotes a vector of objects enrolled to play the specified role R. players.*Rs* denotes the corresponding set.

R(players$[i]$) (where R is a particular role name): An assertion which evaluates to *true* if the ith element of players is enrolled to play the role R, and *false* otherwise. R(players$[i{:}j]$) is *true* if *each* of the elements from the ith through the jth element are enrolled to play R, and *false* otherwise.

Many design patterns require specific methods of specific roles to be invoked in a specific order under various conditions. Thus, for example, in the Observer pattern, when the state of the subject is *modified*, the Update() method of each of the observers must be invoked. Similarly, in the Chain of Responsibility pattern, 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.

Such requirements are conveniently expressed in terms of method *call sequences* or method *call traces*. We will use τ with suitable suffixes as necessary to denote such traces. From an operational perspective, one possibility would be to use a single "global" trace that records *all* of the calls made on all of the objects in the system. But from a reasoning point of view, we will need to reason about the trace of calls made during the execution of particular methods. Thus, each method specification will be expressed in terms of the effects of the method on the variables of its declaring class, as well as the sequence of calls that the method must make during its execution. Note that we associate a separate trace with *each* execution of a method $m()$. This trace will be empty when the execution of $m()$ begins, and will record all of the calls made during that execution; and it must satisfy the requirements imposed by the specification of $m()$.

Consider a call *xx.m*(*args*) that appears in a method $n()$ of a class C. This call will be recorded on the trace τ of the method $n()$ by appending a tuple that records the name of the method invoked, the identity of the object on which the method was invoked, and the arguments passed to the method. The tuple also includes the state of the *invoking* object at the time of the call, and its state at the time of the return. Consider, for example, a scenario in which the state of an object at the time of the return may be different than at the time of the call. This is possible if a method that is invoked, in turn, initiates a chain of calls that result in invocations on the calling object before the original method terminates. In specifying patterns involving such method call sequences, we may need to refer to the state of the invoking object at the time of the call and at the time of the return, and can do so in terms of the appropriate components of the elements within τ. But most patterns do not involve such considerations, and their contracts do not refer to these components. Thus, although our model of traces is such that the trace element corresponding to a given call includes exhaustive information about the call, this complexity is not reflected in the specifications of most common patterns.

The following functions and notations are useful in the expression of trace-based specifications. We should note again that a separate trace τ is associated with each

execution of a given method $n()$; and that each call that $n()$ makes during this execution is represented as a single element in τ.

$|\tau|$**:** Denotes the length of τ (*i.e.*, the number of calls recorded in τ).

$\tau.\,ob$ (where *ob* is an object): Denotes the subsequence formed by projecting out elements in τ corresponding to method invocations on a particular target object *ob*.

$\tau.\,m$ (where $m()$ is a particular method): Denotes the subsequence formed by projecting out elements in τ corresponding to invocations on a particular target method $m()$.

$\tau.\,\sigma o$**:** Denotes the vector of the 'outgoing' states of the invoking object (*i.e.*, the states of the invoking object at the time of each call in τ). Similarly, $\tau.\,\sigma i$ denotes the vector of the 'incoming' states of the invoking object (*i.e.*, the states of the invoking object at the time of the return corresponding to each method call.)

We will often use these functions in combination. Thus, $|\tau.\,ob.\,m|$ denotes the number of invocations recorded in τ to the method $m()$ with *ob* as the target of the invocation. $\tau.\sigma o[k]$ and $\tau.\sigma i[k]$ denote the 'outgoing' and 'incoming' states, respectively, of the invoking object recorded in the kth element of τ. Suppose that τ is the trace corresponding to a method $m()$ of a class C acting on a particular object *ob*. In effect, the change in the state of *ob* from $\tau.\sigma i[k]$ to $\tau.\sigma o[k+1]$ is a result of the actions that $m()$ performs between the calls corresponding to elements k and $k+1$.

One other point concerning traces should be noted. Suppose that in a system built using a pattern P, we have a class C that plays a role R of P. Suppose C contains two methods, $n()$ which does not correspond to a role method of R and $m()$ which does correspond to one of the named methods of R. Suppose further that $m()$ contains a call to $n()$. When specifying $C.m()$, we would include in the trace of $m()$ information about calls it makes to $n()$. But such calls will not be explicitly considered in the corresponding role method specification of $m()$. In other words, the call sequence requirements specified in R's role contract concerning what should be in the trace of $m()$ will generally only concern calls to *named* methods of R. Other methods will be excluded. Thus, when determining whether $C.m()$ meets the requirements specified in the role contract for R, we will exclude from $C.m()$'s trace all of the calls to other methods, and then determine whether the resulting trace satisfies the appropriate call sequence requirements. In some special situations, the role contract may impose general constraints on the *total trace* of $m()$ (*i.e.*, the trace of *all* calls $m()$ makes), including calls to other methods. In such cases, we will use the notation $\tau\tau$ in the role contract to denote the *total trace* of the method in question. Of course, when specifying classes (rather than roles) in our case study, we will be dealing with total traces, and will therefore use $\tau\tau$ in those specifications.

3 Case Study

We now turn our attention to the case study. In Section 3.1, we develop the contract for the Observer pattern. In 3.2, we introduce a typical, if simple, system built using Observer. In 3.3, we develop the subcontract corresponding to the specialization of Observer used in this system. We then demonstrate how a system designer may show that the requirements associated with a given pattern are satisfied, and, most important, how the designer may draw conclusions about the resulting behavior that the system is guaranteed to exhibit. In 3.4, we consider an extended version of the system that uses both the Observer and the Chain of Responsibility patterns, and show how our formalism allows us to reason about the extended system.

3.1 Observer Pattern Contract

The specification of the Observer pattern appears in Figs. 5, 6, and 7. Fig. 5 includes the pattern-level items, Fig. 6 includes the `Subject` role contract, and Fig. 7 includes the `Observer` role contract. Note that for the sake of presentation, we have included the instantiation and invariant clauses in Fig. 5 instead of after the role contracts as required by the contract grammar.

The pattern contract begins with the declaration of two auxiliary concepts. The first concept, *Consistent*, captures the notion of whether a given `subject` state is consistent with a given `observer` state. The second concept, *Modified*, captures what it means for a given `subject` state to be *significantly different* from another state of the `subject`. This concept is required because the `subject` is obliged to update its `observers` whenever its state changes in a manner that is significant to its `observers`. Since which changes in the `subject` state are significant enough to warrant notification depends on the system in question, we use an auxiliary concept to parameterize this notion. In the subcontract, the definitions of these concepts must satisfy the condition stated in the constraints clause, which we will consider after discussing the rest of the contract.

```
pattern Observer contract {
  auxiliary concepts:
    Consistent(Subject, Observer);
    Modified(Subject, Subject);
  constraints:
    ∀s1, s2, o1 :
      [ [¬Modified(s1, s2) ∧ Consistent(s1, o1)]
            ⇒ [Consistent(s2, o1)] ]
  instantiation:
    new Subject()
    lead: newOb;
    pre-cond: true;
    post-cond: (newOb._observers = Φ)
  invariant:
    Subject(players[0])∧
    Observer(players[1 :])∧
    (players.objv[0]._observers =
                players[1:].objs)∧
    (∀ob : (ob ∈ players[1:].objs)) ::
                (ob._subject = players.objv[0])∧
    (∀ob : (ob ∈ players[1:].objs)) ::
                Consistent(players.objv[0], ob))
```

Fig. 5. Observer Contract (part 1)

The *instantiation clause* specifies that a new instance of the pattern is created when an instance of the class playing the Subject role is created. Note that Subject is the lead role for this pattern, as specified in the Subject role contract (Fig. 6). The instantiation clause identifies the newly created object as the lead object, and requires that the _observers field be empty since no observers have yet enrolled to observe this subject.

Next we have the pattern invariant. The first clause of the invariant states that the first object to enroll in an instance of the pattern will play the Subject role. (This of course follows naturally from the instantiation clause.) The next clause states that all other enrolling objects will play the Observer role. The third clause states that the value of the _observers field of the object playing the Subject role will be equal to the set consisting of all of the other enrolled objects [4]. The final clause states that the _subject field of each object playing the Observer role will refer to the object playing the Subject role.

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. This is captured by the final clause of the invariant, which indicates that the states of all of the objects enrolled, with the exception of the lead object, are consistent with the state of the lead object.

```
lead role Subject contract {
  Set _observers;
  void Attach(Observer ob):
    requires: (ob ∉ _observers)
    preserves: ob
    ensures:
      [(_observers = #_observers ∪ {ob})
      ∧¬Modified(#this, this)
      ∧(|τ| = 1) ∧ (|τ.ob.Update| = 1)]
  void Detach(Observer ob):
    requires: (ob ∈ _observers)
    preserves: ob
    ensures:
      [¬Modified(#this, this) ∧ (|τ| = 0)
      ∧(_observers = #_observers − {ob})]
  void Notify( ):
    requires: true
    preserves: _observers
    ensures:
      [¬Modified(#this, this)
      ∧(|τ| = |_observers|)
      ∧∀ob ∈ _observers : (|τ.ob.Update| = 1)]
  others:
    preserves: _observers
    ensures:
      [¬Modified(#this, this) ∧ (|τ| = 0)]
      ∨[(|τ| = 1) ∧ (|τ.this.Notify| = 1)] }
```

Fig. 6. Observer Contract (part 2)

The state of the Subject role, as specified in its contract in Fig. 6, consists of a single field, _observers, whose value will be the set consisting of references to all of the objects currently attached to the subject. The contract does not include an enrollment clause since the only way for an object to enroll as a Subject is by being the lead object in an instantiation of the pattern. Further, the disenrollment clause is omitted since the subject remains enrolled as long as it exists.

The role has three named methods: Attach(), Detach(), and Notify(). Attach() is invoked when an object, ob, wishes to become an observer of the subject. As specified in the requires clause, this method may only be called if ob is not already attached to the subject, and by consequence, not already enrolled in the pattern instance. The preserves clause states that ob is unchanged by the method, and must therefore refer to the same object at the termination of the method as it did at the start of the invocation. The first con-

[4] We should note that players[1:].objs is a mathematical set, whereas players.objv[0]._observers is the value of a variable of the object in question, and is of a programming language type Set. A more precise approach would be to introduce a suitable mathematical model of the object in question in which this variable would be modelled as a mathematical set, and the clause would be written in terms of this model. For ease of presentation, however, we omit such details.

junct of the ensures clause states that the enrolling observer is added to the _observers set[5]. The second conjunct states that the subject will not be *Modified*. Note that this may seem strange since we just stated that the _observers field must be modified. The key, however, is that this change has to do with how the subject keeps track of the objects observing it, and is not the kind of change the observers will likely be interested in. Hence, this kind of change will typically be ignored by definitions of the *Modified*() concept. Indeed, definitions of auxiliary concepts commonly ignore the 'pattern-portion' of the states of the participating objects.

The condition captured by the last two conjuncts of the ensures clause is often overlooked in informal descriptions of the pattern. The key intent of the Observer pattern, as specified by the pattern invariant, is to keep all of the observers *consistent* with the state of the subject. The Attach() method adds a new observer. We must therefore invoke the attaching observer's Update() method, since it might be inconsistent with the state of the subject at the point of the call to Attach(). The first of these two conjuncts states that the length of the method call trace, τ, is 1, indicating that there is only one method call recorded on it. The second conjunct states that this call is to the method Update() on the attaching object.

Detach() is more straightforward. We require that the detaching observer be attached to the subject before the method invocation. When the method terminates, the detaching observer must be removed from _observers, and the subject state must not be modified from its pre-conditional value. Again, the determination of whether the subject's state is modified is based on the definition of *Modifed*() supplied in the relevant subcontract. The call sequence requirement prevents implementations of Detach() from invoking any *named* methods.

The purpose of Notify(), the final Subject role method, is to update each observer with the current state of the subject. As we will see in the others specification, this method must be invoked by any method that *modifies* the state of the subject. The preserves clause requires that _observers not be changed. The first conjunct of the ensures clause states that Notify() must not modify the subject's state. The last two conjuncts require that Update() be invoked on each attached observer. An alternate way to specify this method would be to replace the conditions involving τ with a clause that requires the subject state to be *consistent* with that of the observers. While this is certainly possible, it does not respect the intent of the pattern, which requires Notify to perform the calls to Update() [9]. This is precisely what our contract specifies. If, on the other hand, the pattern did not require such calls, the alternate specification would be appropriate.

The others specification imposes conditions on the methods provided by the class playing the Subject role, outside of the named methods required to support the pattern. These additional methods are required to preserve the _observers set. Further, they must either not modify the state of the subject, or they must invoke the Notify() method. As discussed above, Notify() will in turn invoke Update() on each attached observer.

Note that there is a potential problem with this specification because a careless designer could implement a method that first invokes Notify(), and *then* modifies the state of the subject. Such an implementation would, given that there is a call to Notify(), satisfy our specification, but violate the intent of the pattern. The purpose of the call to Notify() is to ensure that all of the observers are updated to become consistent with the modified state of the subject. If, however, the call to Notify() appears *before* the modification, the observers will not be updated to reflect the post-conditional value of the subject's state. This can be addressed by modifying the second disjunct of the ensures clause as follows:

$$[(|\tau.\mathsf{this.Notify}| \geq 1) \\ \land \neg \mathit{Modified}(\tau.\mathsf{Notify}[\mathsf{last}].\sigma o, \mathsf{this})]$$

The first clause asserts that there is at least one call to Notify() on the current object. The index "last" in the second clause refers to the rightmost element of the given vector. This vector is obtained from τ by retaining only those elements corresponding to invocations of Notify(). "σo" then gives us the state of the object at the time the call was made. Thus, this clause requires that the final state of the subject be *unmodified* from the value that the observers were most recently notified of. For the rest of the discussion, we will ignore this problem and use the simpler ensures clause specified in Fig. 6.

The Observer role contract in Fig. 7 specifies the state of the role as consisting of a single field, _subject. As specified in the pattern invariant, this field holds a reference to the subject object being observed by the observer in question. The enrollment clause states that for an object to enroll in this role, the Attach() method must be invoked on the appropriate subject object, with the enrolling observer passed as argument. The enrollment pre-condition requires that the enrolling observer's _subject field reference the particular subject object it wishes to observe. This, together with the specification of Attach() in Fig. 6, ensures that the con-

[5] Note that when a variable is prefixed with "#" in the ensures clause, it denotes the value the variable had when the method started execution. This is equivalent to the "@pre" notation of OCL [36].

```
role Observer contract {
    Subject _subject;
  enrollment:
    Subject.Attach(ob)
    lead: target
    enrollee: ob
    pre-cond:
      (ob._subject = target) ∧ (source = ob)
    post-cond: true
  disenrollment:
    Subject.Detach(ob)
    lead: target
    disenrollee: ob
    pre-cond: (source = ob)
  void Update():
    requires: true
    preserves: _subject
    ensures: (|τ| = 0) ∧ Consistent(_subject, this)
  others:
    preserves: _subject
    ensures:
     (|τ| = 0) ∧ ∀ss :
    Consistent(ss, #this) ⇒ Consistent(ss, this) } }
```

Fig. 7. Observer Contract (part 3)

ditions specified in the pattern invariant (Fig. 5) will be satisfied when the enrollment action completes, assuming they were satisfied before it began. The clause (source = ob) requires that the object calling the Attach() method be the object enrolling as an observer. In other words, no *third-party* enrollments are permitted by this specification. If we do want to permit such enrollments, this clause must be omitted. The disenrollment clause is similar. Note, however, that there is no need for the clause (ob._subject = target), since the pattern invariant already guarantees this clause, given that ob is enrolled as an observer. The specification of Update() requires that the method preserve the _subject field so that the reference to the subject being observed is not lost. Further, the method must not invoke any named methods, and must leave the state of the observer in a state that is consistent with the state of the subject.

The others specification in Fig. 7 requires that the _subject field be preserved. Further, the ensures clause imposes a call sequence requirement that prevents unnamed methods from invoking any named methods. Finally, if the state of the observer at the start of an invocation is *consistent* with ss, a possible state of the subject, then the state of the observer must be *consistent* with that state at the end of the invocation. The _subject field must be preserved since if it were not, the observer would lose its reference to the subject it is observing. Unnamed methods should not invoke Attach() since the observer must already be attached. Nor should they invoke Notify() since this method must only be invoked by the subject — as we saw in the contract for the Subject role. Detach() is more interesting. Although the specification seems to forbid a call to this method, this is not the case. If an others method were to call Detach(), this object would no longer be enrolled in the pattern at the method's termination, and therefore, the requirements imposed by the ensures clause would not apply. Of course, the call to Detach() would have to satisfy the method pre-condition specified in the Subject role contract, but when control returns to the caller, the object will no longer be enrolled as an observer in the pattern. Hence, the pattern contract's constraints will no longer apply to that object. Thus, unnamed methods can indeed detach observers from their subjects.

The final clause captures a critical aspect of the Observer role's behavior. This clause allows unnamed methods to modify the state of an observer, as long as the modifications do not affect the *consistency* of the observer with respect to its subject. Standard informal descriptions of the pattern suggest that the state of an observer should *not* change except when the Update() method is invoked. But this is unnecessarily restrictive. Suppose, for example, that the observer in question is a type of graphical object that displays information about the state of the subject in the form of a pie-chart. Suppose also that this observer provides a method to switch between *iconified* and *de-iconified* images of the pie-chart. In going from being de-iconified (with the pie-chart information being displayed) to iconified, there is clearly no information lost about the subject. Indeed, we could again de-iconify the observer, and the chart would again be displayed. Therefore, such a change in the state of the observer should be permitted. Our formalism allows for this type of flexibility, given an appropriate definition of the *Consistent*() auxiliary concept in the corresponding subcontract. By contrast, standard informal descriptions of the pattern seem to disallow such changes in the object playing the Observer role[6].

Note that the others specification in Fig. 7 assumes that the state of the subject will not change during the execution of an unnamed method. While this is likely a reasonable requirement for most systems, the pattern is designed to handle situations in which this condition is not satisfied. Suppose, for example, that an unnamed

[6]Once we recognize this, we can of course rewrite the informal description to permit such changes. But the point remains that it was the process of formalization that allowed us to identify this dimension of flexibility that is missing in standard informal descriptions.

method were to invoke some other method on another object, and that this second method were to invoke a method on the original subject. Suppose further that the invocation on the subject resulted in a significant change, as defined by the definition of *Modified*() provided in the corresponding subcontract. Then, as specified in the others clause of the Subject role contract (Fig. 6), Notify() must be invoked, which must in turn call update() on each attached observer — including the observer that initiated the call chain. Thus, when control returns to the original method of the initiating observer, its state will be different from what it was just before the call that initiated the chain. Note, however, that the intent of the pattern will not be violated since the observer's state will be *consistent* with the new state of the subject. To allow for such changes, we must suitably modify the ensures clause of the others specification in Fig. 7. The most interesting modification involves replacing the second conjunct of the ensures clause with the following:

$$
\begin{aligned}
&(|\tau\tau| > 0) \land \\
&\forall \mathsf{ss} : \\
&\quad \mathit{Consistent}(\mathsf{ss}, \#this) \\
&\qquad \Rightarrow \mathit{Consistent}(\mathsf{ss}, \tau\tau.\sigma o[first]) \land \\
&\quad \forall \mathsf{k} : \mathit{Consistent}(\mathsf{ss}, \tau\tau.\sigma i[k]) \\
&\qquad \Rightarrow \mathit{Consistent}(\mathsf{ss}, \tau\tau.\sigma o[k+1]) \land \\
&\quad \mathit{Consistent}(\mathsf{ss}, \tau\tau.\sigma i[last]) \\
&\qquad \Rightarrow \mathit{Consistent}(\mathsf{ss}, this)
\end{aligned}
$$

Recall that $\tau\tau$ denotes the *total trace* of method calls made during a method's execution, including calls to unnamed methods. Recall further that $\tau\tau.\sigma i[k]$ denotes the post-conditional value of the *caller* after the kth method call, and $\tau\tau.\sigma o[k+1]$ denotes the pre-conditional value before the $k+1^{th}$ method call. Hence, this clause requires that if the observer is consistent with some subject state at the point control returns to an unnamed method, the observer must be consistent with that same subject state at the point of the next outgoing method call. Similar conditions are imposed relating the pre-conditional state of the observer and the first method call, and the post-conditional state of the observer and the last method call. Note that the clause is expressed using the total trace because it must handle changes in state that are initiated *indirectly* through unnamed methods — as in the example presented above. Note further that in the case where $\tau\tau$ is empty, we must again require the same consistency condition specified in Fig. 7. It is interesting to note that our contract formalism brings to the surface such subtle issues that are easily missed in informal descriptions.

Let us now turn our attention to the *constraints* imposed on auxiliary concept definitions in the pattern contract (Fig. 5). For convenience, we reproduce the constraints clause here.

$$
\begin{aligned}
&\forall s1, s2, o1 : \\
&\quad [\, [\neg \mathit{Modified}(s1,\ s2)\ \land \mathit{Consistent}(s1,\ o1)] \\
&\qquad \Rightarrow [\mathit{Consistent}(s2,\ o1)]\,]
\end{aligned}
$$

Suppose that a given subject state *s1* is *consistent* with a given observer state o1 according to the definition of *Consistent*() provided by the appropriate subcontract. Suppose the state of the subject changes to *s2*, and that *Modified*(*s1*, *s2*) evaluates to false according to the definition of *Modified*() provided in the same subcontract. Finally, suppose that the new state *s2* is *not consistent* with the observer state *o1*, according to the same definition of *Consistent*(). At this point, the intent of the pattern will be violated. The subject will not update its observers, since in its view, as captured in the definition of *Modified*(), its state has not changed in a significant way. According to the definition of *Consistent*(), however, the observer's state *o1* is not consistent with the new state of the subject. The problem arises here not because of the failure of the subject or the observers to interact in a manner intended by the pattern, but rather because of mutually incompatible notions of *consistency* and *significant modification*. The concept constraints in our pattern contracts serve to eliminate such incompatibilities. Thus, while the definitions of a pattern's auxiliary concepts may be tailored to the needs of a particular system, the concept constraints in the contract ensure that this flexibility is not used in a way that would violate the intent of the pattern.

3.2 SimSickCity System

SimSickCity, the system used in our case study, implements a primitive simulation of a hospital. The implementation consists of three main classes: Patient, Nurse, and Doctor. Instances of these classes represent the corresponding entities in the simulated hospital. We first consider the Patient class, shown in Figs. 8 and 9. Some of the code has been elided for the sake of presentation.

The constructor receives a string parameter containing the name of the patient object being constructed. First, it initializes temp and hrtRt to suitable values, corresponding to the patient's temperature and heart rate, respectively. Next, it initializes _nurses, the set of references to the nurse objects assigned to this patient, to the empty set. Finally, it sets _doc, the reference to the doctor object[7] for this patient, to null. The ensures clause expresses this behavior. It also states that the constructor does not make any calls to any other methods during its execution. As we noted in Section 2.3, we use

[7] At most one doctor object may be assigned to a patient object at any time. Any number of nurse objects may be assigned to a patient.

$\tau\tau$ here to denote the *total trace*, which includes *all* of the calls made by the method — not just those to named methods.

```
public class Patient {
  private String name;
  private int temp, hrtRt;
  private Set _nurses;
  private Doctor _doc;
  public Patient(String n) {
    /* initialize name, temp, hrtRt */
    /* set _doc to null, _nurses to empty set */ }
    requires: true
    ensures: [(name = n) ∧ (_nurses = Φ)
      ∧(_doc = null) ∧ (ττ = ⟨⟩)]
  public int tempInfo() { return temp; }
    requires: true
    preserves: name, temp, hrtRt, _nurses, _doc
    ensures: [(result = temp) ∧ (ττ = ⟨⟩)]
  public int heartRateInfo() { return hrtRt; }
    requires: true
    preserves: name, temp, hrtRt, _nurses, _doc
    ensures: [(result = hrtRt) ∧ (ττ = ⟨⟩)]
  public String pname() { return name; }
    requires, etc.: similar; details omitted
  public void addNurse(Nurse n)
    { /* add n to _nurses, n.update() */ }
    requires: (n ∉ _nurses)
    preserves: name, temp, hrtRt, _doc
    ensures: [(_nurses = #_nurses ∪ {n})
      ∧(|ττ| = |ττ.n.update| = 1)]
  public void removeNurse(Nurse n)
    { /* remove n from _nurses */ }
    requires: (n ∈ _nurses)
    preserves: name, temp, hrtRt, _doc
    ensures:
      [(_nurses = #_nurses − {n}) ∧ (ττ = ⟨⟩)]
  public void assignDoctor(Doctor d)
    { /* set _doc to d */ }
    requires: (_doc = null)
    preserves: name, temp, hrtRt, _nurses
    ensures:
      [(_doc = d) ∧ (|ττ| = |ττ.d.update| = 1)]
  public void deassignDoctor( )
    { /* set _doc to null */ }
    requires: (_doc ≠ null)
    preserves: name, temp, hrtRt, _nurses
    ensures: [(_doc = null) ∧ (ττ = ⟨⟩)]
```

Fig. 8. Patient Class (part 1)

The methods tempInfo() and heartRateInfo() return the temperature and heart rate information about the patient, respectively. These methods do not make any changes in the state of the patient, nor do they call any methods. The pname() method is defined similarly. The **preserves** and **ensures** clauses express this behavior. addNurse(n) adds n to the set of nurses assigned to the Patient, and calls n.update(). This call is required to ensure that the state of the nurse being added to the patient's _nurses set is consistent with the state of the patient at the method's termination. The **requires** clause states that n must not be contained in _nurses; i.e., n must not already be assigned to the patient. The **ensures** clause asserts that n is added to _nurses, and that the only method call made by addNurse() is to n.update(). removeNurse(n) removes n from the set of nurses assigned to the Patient, provided n is currently assigned to the patient. assignDoctor(d) and deassignDoctor() are defined similarly.

```
  public void changeCondition( ) {
    /* set temp, hrtRt to new values *
    Notify(); }
    requires: true
    preserves: name, _nurses, _doc
    ensures: (|ττ| = 1) ∧ (|ττ.this.Notify| = 1)
  public void Notify( ) {
    /* call n.update(this) for each n in _nurses */
    /* call _doc.update(this) if _doc is not null */ }
    requires: true
    preserves: temp, hrtRt, name, _nurses, _doc
    ensures: [(|ττ| = |_nurses| + |_doc|) ∧
      (∀n ∈ _nurses : (|ττ.n.update| = 1)) ∧
      ((_doc ≠ null) ⇒ (|ττ._doc.update| = 1))]
```

Fig. 9. Patient Class (part 2)

The method changeCondition(), shown in Fig. 9, simulates a change in the condition of the patient. This method randomly sets temp and hrtRt to new values, and then invokes the patient's Notify() method. The **ensures** clause specifies this.

Notify() invokes the update() method on each Nurse object in the _nurses set. It also invokes the update() method on the _doc object, provided that this field is not set to null. In the **ensures** clause of the method, we use the notation |_doc| to denote 0 if _doc is null, and to denote 1 otherwise. Thus, the first portion of the **ensures** clause asserts that the number of calls made during the execution of Notify() is equal to the number of elements in _nurses plus 0 or 1, depending on whether _doc is set to null. The remaining two clauses state that update() is called on each element of _nurses, and on _doc if the field is not set to null. Note that these clauses do not constrain the *order* in which

these calls are made. Thus it is possible that update() will be invoked on some of the nurse objects, followed by the invocation on _doc, followed by the invocations on the remaining nurses.

```
public class Nurse {
  private HashMap pInfo; /* info about patients */
  public Nurse() {
    /* initialize pInfo to new empty map */ }
    requires: true
    ensures: [(pInfo = Φm) ∧ (ττ = ⟨⟩)]
  public void watch(Patient p){
    pInfo.put(p, null); p.addNurse(this); }
    requires: (p ∉ pInfo.keySet)
    ensures: [((|ττ| = 1 = |ττ.p.addNurse|)
      ∧(ττ[0].arg = this)
      ∧(ττ[0].pInfoO = #pInfo ⊕ (p, null))
      ∧(pInfo = ττ[0].pInfoI) ]
  public void ignore(Patient p){
    pInfo.remove(p); p.removeNurse(this); }
    requires: (p ∈ pInfo.keySet)
    ensures: [(pInfo = #pInfo ⊖ p)
      ∧(|ττ| = 1 = |ττ.p.removeNurse|)
      ∧(ττ[0].arg = this)]
  public void update(Patient p){
    int t = p.tempInfo();
    pInfo.put(p,new Integer(t)); }
    requires: (p ∈ pInfo.keySet)
    ensures: [(pInfo = #pInfo ⊕ (p, p.temp))
      ∧(|ττ| = 1 = |ττ.p.tempInfo|)]
  public string pStatus(Patient p) {
    int t = ((Integer)pInfo.get(p)).intValue();
    if (t<105) { return("good"); }
    else { return("bad"); } }
    requires: (p ∈ pInfo.keySet)
    preserves: pInfo
    ensures: [(ττ = ⟨⟩)
       ∧((pInfo[p] < 105) ⇒ (result = "good"))
       ∧((pInfo[p] ≥ 105) ⇒ (result = "bad"))]
```

Fig. 10. Nurse Class

Let us next consider the Nurse class shown in Fig. 10. Information about the patients being watched by a given nurse is stored in pInfo, a HashMap indexed by the Patient objects. The constructor initializes pInfo to the empty hash map, (Φ_m), and does not invoke any methods during its execution. The watch(p) method is used to add a Patient object, p, to the set of patients being watched by a nurse, provided p is not already being watched[8]. watch(p) adds p with an empty associated value to pInfo, and then invokes p.addNurse(). addNurse(), as specified in Fig. 8, will in turn invoke update() on the Nurse object, passing the new Patient object being watched as an argument. update(), as we will discuss shortly, will refresh the current information about the patient stored in pInfo. The **ensures** clause for watch() first states that the addNurse() method will be called with the appropriate argument. It next states that the value of pInfo at the point of the call to addNurse() will be the same as its value at the start of the watch() method, except for the addition of p to pInfo.keySet. (pInfoO denotes the *outgoing* value of pInfo.) Finally, it states that the post-conditional value of pInfo will be the same as its value when addNurse() returns. (pInfoI denotes the *incoming* value of pInfo.) We use the notation "⊕" in the **ensures** clause to denote the addition of the specified pair to the hash map.

The ignore(p) method is used to remove p from the set of patients being watched by a given nurse. The method removes p from pInfo, and invokes removeNurse() on p, passing the nurse in question as the argument. These properties are expressed in the **ensures** clause. The "⊖" notation denotes removal of the element that has p as the key from the hash map.

There is an important question that must be considered here. According to the specification of the Patient class, p.addNurse() requires, as part of its precondition, that the nurse being added not already be in p._nurses. How can we be sure, at the point of the call to p.addNurse(this) from within Nurse.watch(), that this condition is satisfied? There is a similar question concerning the call to p.removeNurse() from within ignore(p) since this call requires that the nurse being removed be contained in p._nurses. In general, in systems that consist of objects that hold references to other objects which are not *encapsulated*, a suitable invariant is required that expresses the *reference relations* among the objects that can be used to show that such requirements are met. We will refer to this invariant as a reference-relation-invariant (*RRI*). The *RRI* for our system is:

$$[(p \in n.\text{pInfo.keySet} \Leftrightarrow n \in p.\text{_nurses}) \wedge (p \in d.\text{pInfo.keySet} \Leftrightarrow (d = p.\text{_doc}))] \qquad (RRI)$$

where p, n, d denote a Patient, Nurse, and Doctor object, respectively. Establishing this invariant formally requires system-wide reasoning. Since our interest is in reasoning about design patterns, rather than general aliasing issues, we present an informal argument to justify its validity. Note first that the invariant is satisfied trivially at the point of object creation. Next, note that Patient._nurses is modified only by the addNurse() and removeNurse() methods of Patient, and Patient._doc is modified only

[8] keySet() is a method provided by the HashMap collection of *Java* that returns the set of "keys" currently in the hash map. We use keySet (without "()") in assertions to refer to this set.

by assignDoctor() and deassignDoctor(). The former methods are invoked only by the watch() and ignore() methods of the Nurse class, and the latter are only invoked by the watch() and ignore() methods of the Doctor class[9]. Thus, given the pre-conditions of the watch() and ignore() methods, and the actions performed by these methods (as expressed in their post-conditions), we can see that the pre-conditions of the addNurse()/assignDoctor() and removeNurse()/deassignDoctor() methods will be met when those methods are invoked. Given the actions performed by these various methods, the invariant will be satisfied when control leaves these methods.

Next consider the update() method of the Nurse class. This method uses Patient.tempInfo() to obtain the temperature of the patient object passed as argument, and uses this value to refresh the information stored in pInfo about the patient. Note that we use the "⊕" notation in the **ensures** clause to indicate that the patient information stored in pInfo will be *replaced* by the new information.

The final method, pStatus(p), returns the "status" of p as being "good" or "bad", depending on whether the temperature recorded for p in pInfo is less than 105. We should stress that even if this method returns "good", we cannot be sure that the *current* temperature of p is, in fact, less than 105. All we can be sure of, based on the specifications given in the Nurse class, is that the temperature recorded for p within pInfo matches the temperature of p at the point update() was last called with p passed as argument. But since the state of p might have changed since that point, the information stored within pInfo about the patient might be stale. That this is *not* the case, that the information stored within pInfo is indeed current, depends on the system being built according to the Observer pattern. Showing this fact is the goal of the reasoning process that we present in the next subsection. Following that reasoning, we will be able to conclude that if the pStatus(p) method returns the result "good", then the value of temp in the *current* state of the patient p must in fact be less than 105. More abstractly, we will be able to conclude, following the reasoning in the next subsection, that the information stored within each nurse object about the state of each patient object that it is watching is *consistent* with the current state of the patient.

The Doctor class shown in Fig. 11 is defined analogously to the Nurse class. The only difference is that instances of Doctor record information returned by the

```
public class Doctor {
  private HashMap pInfo; /* info about patients */
  public Doctor() {
    /* initialize pInfo to new empty map */ }
    requires: true
    ensures: [(pInfo = Φm) ∧ (ττ = ⟨⟩)]
  public void watch(Patient p){
    pInfo.put(p, null); p.assignDoctor(this); }
    requires: (p ∉ pInfo.keySet)
    ensures: [(|ττ| = 1 = |ττ.p.assignDoctor|)
      ∧(ττ[0].arg = this)
      ∧(ττ[0].pInfoO = #pInfo ⊕ (p, null))
      ∧(pInfo = ττ[0].pInfoI) ]
  public void ignore(Patient p){
    pInfo.remove(p); p.deassignDoctor(this); }
    requires: (p ∈ pInfo.keySet)
    ensures: [(pInfo = #pInfo ⊖ p)
      ∧(|ττ| = 1 = |ττ.p.deassignDoctor|)
      ∧(ττ[0].arg = this)]
  public void update(Patient p){
    int t = p.heartRateInfo();
    pInfo.put(p, new Integer(t));
    requires: (p ∈ pInfo.keySet)
    ensures: [(pInfo = #pInfo ⊕ (p, p.hrtRt))
      ∧(|ττ| = 1 = |ττ.p.heartRateInfo|)]
  public string pStatus(Patient p) {
    int t = ((Integer) pInfo.get(p)).intValue();
    if (t>0) { return("good"); }
    else { return("bad"); } }
    requires: (p ∈ pInfo.keySet)
    preserves: pInfo
    ensures: [(ττ = ⟨⟩)
       ∧((pInfo[p] > 0) ⇒ (result = "good"))
       ∧((pInfo[p] ≤ 0) ⇒ (result = "bad"))]
```

Fig. 11. Doctor Class

Patient.heartRateInfo() method rather than by the Patient.tempInfo() method. As in the case of nurse objects, we cannot be sure that a given doctor object has the correct *heart rate* information about each of the patient objects that it is currently watching. That conclusion can be arrived at only after we perform the reasoning process described in the next subsection to show that the requirements of the Observer pattern are met.

3.3 Reasoning About Patterns in SimSickCity

In reasoning about the patterns used in a system, our first task is to specify the subcontracts corresponding to the pattern specializations. Next, given the mappings specified in the subcontracts, we must check that all of the requirements specified in the pattern contracts are

[9] A natural question to ask at this point is, how do we know that these methods are only invoked from the specified locations? This is one of the key points that would have to be established during system-wide reasoning.

satisfied. Finally, we can appeal to the invariants provided in the pattern contracts, as specialized by the corresponding subcontracts, to arrive at conclusions about the system's behavior.

The subcontract for the PND specialization (for "Patient, Nurse, Doctor") appears in Figs. 12 and 13. First

```
specialization PND : Observer subcontract {
  rolemap Patient as Subject {
    lead relation: (lead = this)
    state: _observers = {
      Set obss = new HashSet(_nurses);
      if (_doc != null) { obss.add(_doc); }
      return (obss); }
    methods:
      Attach(Observer x) :
        (Nurse?(x):addNurse(x) ||
        Doctor?(x):assignDoctor(x))
      Detach(Observer x) :
        (Nurse?(x):removeNurse(x) ||
        Doctor?(x):deassignDoctor(){x=_doc})
      Notify() : notify() }
```

Fig. 12. Subcontract for PND (part 1)

we have the role map corresponding to the Subject role, which is played by the Patient class. The lead relation clause states that an instance of the Patient class plays the lead role in any instance of PND. The state component specifies that the _observers field of the Subject role is played by a combination of _nurses and _doc. The state map tells us that the set of observers of any Patient object is the set of Nurse objects currently watching it, plus the Doctor object, if any, assigned to it.

Next, we have the method maps. Since, as specified in the rest of the subcontract, both Nurse and Doctor play the Observer role, and the argument to Attach() is an Observer, we have to specify two mappings for this method. The first states that Attach() with a nurse as argument is mapped to addNurse() with the same argument. The second mapping states that if the argument to Attach() is a doctor, the method is mapped to assignDoctor() with the same argument. Note that in both cases, there are no further argument maps to specify. Detach() is similar except in the case that the argument is a doctor. In that case, the method is mapped to deassignDoctor() (which has no arguments) with the argument to Detach() being, as specified in the argument map, the same as the doctor object referenced by _doc. The Notify() method of the Subject role maps to the method of the same name in Patient.

The role map for Nurse as an Observer appears in Fig. 13. The lead relation states that given a particular

```
rolemap Nurse as Observer {
  lead relation: (lead ∈ pInfo.keySet)
  state: _subject = { lead
  methods:
    Update(): update(lead) }
rolemap Doctor as Observer {
  /* identical to that for Nurse as Observer */ }
auxiliary concepts:
  Modified (Patient p1, Patient p2) {
    return ((p1.temp != p2.temp) ||
          (p1.hrtRate != p2.hrtRate)); }
  Consistent (Patient p1, Nurse n1) {
    int n1temp =
    ((Integer) n1.pInfo.get(lead)).intValue();
    return (p1.temp == n1temp); }
  Consistent(Patient p1, Doctor d1) {
    int d1hrtRate =
    ((Integer) d1.pInfo.get(lead)).intValue();
    return (p1.hrtRate == d1hrtRate); } }
```

Fig. 13. Subcontract for PND (part 2)

pattern instance, the patient playing the lead role in that instance will be in the set of keys defined within pInfo. The state map is required to specify a mapping for the _subject field of the Observer role (Fig. 7). When a nurse object is viewed as an observer in the context of a particular pattern instance, its _subject field will reference the object playing the lead role for that instance (*i.e.*, the observer's subject). Hence, this field is mapped to the *lead* object for the pattern instance under consideration. Nurse.update() maps directly to the Update() method of the Observer role. The argument to Nurse.update() is identified as the lead object — the object used to determine the pattern instance under consideration when reasoning about a call to Nurse.update(). The role map for Doctor as an Observer is identical.

The last part of the subcontract specifies definitions for the auxiliary concepts declared in the Observer pattern contract (Fig. 5). As specified in the contract, *Modified*() takes two subject states as argument. In the PND specialization, only Patient plays the Subject role. Therefore, we need only one definition for this concept. The definition specifies that the state p2 is *modified* from p1, and hence *Modified*() should return true, if either the temp values in the two states or the hrtRate values in the two states are different from each other.

Two definitions are required for the *Consistent*() concept since one of the arguments to this concept is an observer, which can be either a nurse or a doctor ob-

ject. In both cases the definitions are straightforward. In the first case, we define a patient state p1 to be consistent with a nurse state n1 if the value of p1.temp agrees with the value recorded in n1.pInfo for the patient playing the lead role in the pattern instance. The definition of *Consistent*() when the argument is a doctor, is similar.

The next step in our reasoning process is to check, given the PND subcontract and the specifications of the Patient, Nurse and Doctor classes, that the requirements specified in the Observer pattern contract are satisfied. We will consider the requirements in the order that they are specified in the pattern contract (Figs. 5, 6, 7). First, recall the constraint that definitions of *Modified*() and *Consistent*() are required to satisfy:

$$\forall s1, s2, o1 : [\,[\neg \mathit{Modified}(\mathsf{s1}, \mathsf{s2}) \wedge \mathit{Consistent}(\mathsf{s1}, \mathsf{o1})] \Rightarrow \mathit{Consistent}(\mathsf{s2}, \mathsf{o1})]$$

Since we are considering instances of the PND specialization, s1 and s2 must be states of a patient object. Hence, from the definition of *Modified*(), it follows that for the antecedent to be true, s1.temp and s1.hrtRate must be equal to the corresponding fields in s2. Further, we know that o1 must be either the state of a nurse object or the state of a doctor object. Suppose o1 is a nurse. From the first definition of *Consistent*() in Fig. 13, we can conclude that for the antecedent to be true, the temp value recorded for this patient in o1 must be equal to s1.temp; hence it must also be equal to s2.temp. We can therefore conclude that *Consistent*(s2, o1) must also be satisfied. The case when o1 is a doctor object is similar.

Next consider the pattern instantiation condition specified in Fig. 5. From the ensures clause of the Patient constructor in Fig. 8, and the role map specified in Fig. 12, we can see that the value of _observers will be empty when a patient is created, as required by the instantiation condition.

Next consider the conditions that must be satisfied by the *named* methods of Subject, as specified in Fig. 6. According to the PND subcontract shown in Fig. 12, there are two methods that map to Subject.Attach(): Patient.addNurse() and Patient.assignDoctor(). Given the definition of *Modified*() and the mapping of _observers, it is easy to check that these methods satisfy the conditions imposed on Attach() by the pattern contract. In particular, note that the call sequence requirement that requires the Update() method to be invoked on the attaching observer is satisfied. removeNurse() and deassignDoctor() can similarly be seen to satisfy the requirements imposed on Detach() by the pattern contract.

The remaining named method of Subject is Notify(). Given the field mapping defined in Fig. 12 that maps the set of nurse objects in Patient._nurses, plus the doctor object, if any, referenced by Patient._doc, to Subject._observers, we can see that Patient.Notify() makes the calls required of it by the Subject role contract in Fig. 6.

Next consider the *other* methods: tempInfo(), heartRateInfo(), pname(), and changeCondition(). These methods are required to satisfy the others specification shown in Fig. 6. For the first three methods, this is immediate since they do not change the state of the Patient, and do not call any named methods. Patient.changeCondition(), however, causes changes in the state of the system. According to its specification in Fig. 9, it invokes Notify() on the patient in question, and hence also satisfies the others specification.

Before turning to the classes playing the Observer role, let us consider a possible change in the Patient class. Suppose we wished to keep track of the number of nurses assigned to watch each patient. For this, we could introduce a suitable field in the Patient class, call it _noOfNurses, initialize it to 0 in the constructor, and increment and decrement it by 1 in addNurse() and removeNurse(), respectively. We could also add a *get* method that returns the number of nurses currently watching the patient. With these changes, we could consider revising the definition of *Modified*() in Fig. 6 to say that p1 is different from p2 if the value of _noOfNurses in p1 is different from the corresponding value in p2 — even if the values of temp and hrtRate in the two states are the same. We could similarly revise the definition of *Consistent*() to say that p1 is consistent with o1 if and only if the values of p1._noOfNurses, p1.temp, and p1.hrtRate are equal to the corresponding values stored in o1 about p1. This might reflect the fact that the hospital system considers the number of nurses assigned to each patient as a critical part of the patient's state. Each nurse might, for example, need to cross-check patient medications with those administered by other nurses. But if we did this, it would violate the intent of the pattern! The addNurse() method, for example, would not satisfy the conditions imposed on Attach(); when the method finishes, the state of the subject will be *modified* according to the new concept definition. Further, since the original implementation of addNurse() does not invoke Notify(), previously attached nurse objects will be left in states that are *inconsistent* with the patient. These are the kinds of changes that might be made during system evolution. In making such changes, the system designers/maintainers need to ensure that the system continues to be *faithful* to the patterns used in its design. Our formalism, by pinpointing

such problems, can be of considerable help in doing so. We will return to this point in the final section.

Let us now consider the Nurse class to see if it meets the requirements of the Observer role. The Update() method of the role is mapped, according to the Nurse role map specified in Fig. 13, to Nurse.update(). From the specification of Nurse.update() in Fig. 10, we can conclude that this method will update the temp value in Nurse.pInfo associated with the patient passed as argument to reflect the patient's current temp value. The specification also states that the only call made during the execution of Nurse.update() is to Patient.tempInfo(). As discussed previously, this method does not *modify* the patient's state. Thus, given the definition of *Consistent*() specified in Fig. 13, for the case of the observer being a nurse, we can conclude that Nurse.update() meets the requirements specified in Fig. 7 for the Update() method of the Observer role.

The watch(p) method of Nurse poses an interesting problem. According to the pre-condition and the *reference-relation-invariant* (*RRI*), when the method begins executing, the nurse in question is not observing p. Hence, the nurse is not enrolled in the instance of PND for which p is the lead object. But by the time watch() terminates, the nurse *will be* enrolled according to the post-condition of the method and the Observer enrollment clause specified in Fig. 7. Therefore, we must require the portion of watch() following the completion of the enrollment action to satisfy the condition imposed by the Observer role contract on *other* methods. This requirement is satisfied since according to the post-condition of watch(), the post-conditional value of pInfo will be the same as its value when the call to addNurse() — which is the enrollment action — finishes and returns to watch(). Hence, the final state of the nurse will be consistent with any patient state with which the nurse state was consistent immediately following the enrollment action. Note that this is an important requirement since in its absence, we could arbitrarily change the value recorded in pInfo for the temperature of p following the return from the call to addNurse(). This would obviously violate the intent of the pattern in the same manner as if we were to introduce such a change in pStatus(). The reasoning steps required to check that the Doctor class meets the conditions specified in the Observer role contract are analogous; we omit the details.

The final step in our reasoning process is to appeal to the pattern invariant in Fig. 5, making the appropriate substitutions based on the mappings specified in the PND subcontract. The most interesting part of the pattern invariant is its fifth clause:

$$(\forall \mathsf{ob} : (\mathsf{ob} \in \mathsf{players[1:].objs})) :: \quad \mathit{Consistent}(\mathsf{players.objv}[0], \mathsf{ob})) \qquad \text{(PI)}$$

The remaining clauses in the invariant specify additional properties. First, the invariant specifies that the first object to enroll, players[0], plays the roll of Subject. Hence, given the PND subcontract, players[0] is the patient object playing the lead role. Second, the invariant specifies that all additional enrolled objects, players[1:], are enrolled as observers. Again, given the PND subcontract, these objects are the nurse objects and/or doctor object assigned to the patient. The invariant also specifies that the _observers field of the subject contains the set of objects enrolled as observers, and that the _subject field of each observer is a reference to the subject. This information, given the mappings of these fields in the PND subcontract, and the lead relations in the subcontract, can be combined with (PI) to derive the following (n being of type Nurse, p being of type Patient and d being of type Doctor):

$$\begin{aligned}[&(\mathsf{n} \in \mathsf{p._nurses}) \Rightarrow \\ &\quad ((\mathsf{p} \in \mathsf{n.pInfo.keySet}) \wedge \mathit{Consistent}(\mathsf{p}, \mathsf{n})) \\ \wedge\, &(\mathsf{d} = \mathsf{p._doc}) \Rightarrow \\ &\quad ((\mathsf{p} \in \mathsf{d.pInfo.keySet}) \wedge \mathit{Consistent}(\mathsf{p}, \mathsf{d}))\,]\end{aligned}$$

Finally, substituting the definitions of *Consistent*() provided in the PND subcontract, this simplifies to:

$$\begin{aligned}[&(\mathsf{n} \in \mathsf{p._nurses}) \Rightarrow \\ &\quad ((\mathsf{p} \in \mathsf{n.pInfo.keySet}) \wedge (\mathsf{n.info[p]} = \mathsf{p.temp})) \\ \wedge\, &(\mathsf{d} = \mathsf{p._doc}) \Rightarrow \\ &\quad ((\mathsf{p} \in \mathsf{d.pInfo.keySet})) \wedge (\mathsf{n.info[p]} = \mathsf{p.hrtRt})\,]\end{aligned}$$

The first implication states that if n is a nurse object observing p, then n will contain information about p, and the information will reflect the actual current state of p. The second implication gives us a similar assurance if d is a doctor object assigned to watch p. These two implications capture the essential *pattern-centric* behavior of the SimSickCity system. Their validity depends on the various classes correctly playing the roles of the Observer pattern. Correspondingly, in our reasoning, we could not establish these results based solely on the specifications of the individual classes. Instead, we had to first establish that the requirements expressed in the pattern contract were met based on the mappings specified in the subcontract, and then appeal to the invariant guaranteed by the pattern contract to arrive at the expected results.

Before concluding this section, it is worth mentioning that the standard informal description of the Observer pattern [9] includes a Subject.getState() method. Classes playing the Observer role invoke this method from within Observer.Update() to access the information necessary to update their state. While this is one way to achieve the desired behavior, it is not re-

quired to satisfy the pattern's intent. In general, the portions of the subject state that are relevant to a particular observer will vary on a per observer basis. Requiring each observer to get the entire state of the subject is unnecessary. In our case study, for example, a nurse observer is only interested in the temperature of the patient subject, while a doctor observer is interested only in the heart rate of the patient. Our pattern contract makes it clear that Observer.Update() is responsible for bringing the observer into a state that is consistent with the subject. How this is done, in particular what information about the subject's state is needed to do this, is entirely up to the application. Indeed, this may vary from one observer to another — as in our case study. The relevant details are captured in the subcontract corresponding to the particular specialization. This again illustrates how the process of developing pattern contracts enables us to identify dimensions of flexibility that may be missing in informal descriptions.

3.4 An Extended System

Consider again the Nurse class. Suppose that in some part of the SimSickCity system it is necessary to find the status of a particular patient object p. If we knew that a particular nurse object n was watching p, we could get this information by calling n.pStatus(p). Suppose, however, that we do not have information about which nurse objects are assigned to which patient objects. In that case, one approach would be to arrange the nurse objects in a *chain*, and to apply the *Chain of Responsibility* (*CoR*) pattern [9][10].

The main behavioral requirement of the CoR specifies that when an object in the chain receives a *request*, it *handles* the request if it is able to do so, or if it is not, passes it on to its *successor* in the chain. The last object in the chain, if it is not able to properly handle the request, performs some default action since it does not have a successor to pass the request to.

The class in Fig. 14 is a simple variation on the original Nurse class that chains nurse objects together during construction. The _next field is used to maintain a reference to the next nurse object in the chain. If there is no successor, _next will be set to null. The main behavioral modification is in the pStatus() method. This method returns the same information about p as it did previously, provided the patient is being watched by the nurse. If not, the method forwards the status request to the nurse's successor, assuming such an object exists. If there is no successor, a problem in the chain will be detected, and a suitable message will be returned. The revised specification of pStatus() captures this behavior. In particular, note that the third implication of the ensures clause states that if the request is forwarded to the successor, the result returned by the original call will be equal to the result returned by the forwarded call.

[10] There are of course other alternatives. Since, however, our interest is in using patterns, we focus on this approach exclusively.

```
public class Nurse {
  private HashMap pInfo; /* info about patients */
  private Nurse _next; /* successor in CoR */
  public Nurse() {
    _next = null;
    /* initialize pInfo to new empty map */ }
    requires: true
    ensures:
      [(pInfo = Φm) ∧ (ττ = ⟨⟩) ∧ (_next = null)]
  public Nurse(Nurse n) {
    _next = n;
    /* initialize pInfo to new empty map */ }
    requires: (n ≠ null)
    ensures:
      [(pInfo = Φm) ∧ (ττ = ⟨⟩) ∧ (_next = n)]
  /* ... other methods as before... */
  public String pStatus(Patient p) {
    if (pInfo.keySet().contains(p)) {
      int t = ((Integer) pInfo.get(p)).intValue();
      if (t<105) { return("good"); }
      else { return("bad"); } }
    if (_next! =null) { return(_next.pStatus(p)); }
    else { return("Problem in chain!"); } }
    requires: true
    preserves: pInfo, _next
    ensures:
      [(p ∈ pInfo.keySet) ⇒ (original ensures)]
    ∧ [((p ∉ pInfo.keySet) ∧ (_next = null)) ⇒
        ((ττ=⟨⟩)∧(result="Problem in chain!"))]
    ∧ [((p ∉ pInfo.keySet) ∧ (_next ≠ null)) ⇒
        ((|ττ| = 1) ∧ (|ττ._next.pStatus| = 1)
        ∧(ττ[0].args = p)
        ∧(result = ττ[0].result))]
```

Fig. 14. Nurse Class (extended)

Note that *cycles* are not possible in the chain of nurse objects since the _next link of each nurse is set when the objects are constructed. It is, however, possible to have multiple nurse objects share a successor. Thus, the structure could be a *tree*, rather than a chain. For convenience, we assume that such structures are disallowed, and that all of the nurse objects are arranged in a linear chain. We further assume that the main() method defines a variable hn (for 'head nurse') that references the first object in the chain. All pStatus() requests are sent to hn.

The key behavioral property exhibited by this system is that if p is a patient object and p._nurses is not empty, then the call hn.pStatus(p) will return the current status of p. It is worth stressing that the validity of this property depends on both patterns being applied correctly. The CoR pattern, in conjunction with the assumption that all nurse objects in the system are in the chain, starting with the one referenced by hn[11], and the assumption that p._nurses is not empty, allows us to conclude that a legal patient status will be returned by the head nurse object. The *Observer* pattern, as we saw in Section 3.3, ensures that if a nurse object returns the result "good" or "bad", the result must indeed reflect the actual *current* status of the patient in question. Together, these two results establish the behavioral property of interest. To complete the reasoning process, we must show that the CoR pattern has been implemented correctly. To do this, we must consider the pattern contract, and the subcontract corresponding to this particular application. In the rest of this section, we sketch the broad outline of this reasoning task.

```
pattern Chain of Responsibility contract {
  auxiliary concepts:
    CanHandle(Handler, Request);
  role Request contract { }
  role Response contract { }
  lead role Handler contract {
    Handler _succ;
    Response handle(Request r):
      requires:
        ¬CanHandle(this, r) ⇒ (_succ ≠ null)
      preserves: _succ, r
      ensures:
       [CanHandle(this, r) ⇒ (τ = ⟨⟩)]∧
       [¬CanHandle(this, r) ⇒
         ((|τ| = 1) ∧ (|τ[0]._succ.handle| = 1)∧
         (τ[0].args = r) ∧ (result = τ[0].result))]
  }
  invariant result:
  ∀h, r, re :
   [(re = h.handle(r)) ⇒
     ∃h_1, ..., h_n : [(h = h_1)∧
       ∀i < n : [(h_i._succ = h_{i+1})
            ∧¬CanHandle(h_i, r) ∧ CanHandle(h_n, r)
            ∧ (re = h_n.handle(r))]]]
}
```

Fig. 15. Chain of Responsibility Contract

[11]Note that this is essentially a set of *reference relations* about the nurse objects in the system. Thus, in a more complete presentation, we would express this by adding appropriate clauses to *RRI*, our reference-relation-invariant.

Key portions of the pattern contract for the *CoR* pattern are shown in Fig. 15. The Handler role is the primary role in the pattern; each object in the handling chain plays this role. The other two roles, Request and Response, as their names suggest, represent a *request* to a handler and the *response* that the handler returns. These roles have no named methods or state components, and serve only to identify the type of argument passed to the handler method and the corresponding return type. There is one auxiliary concept, *CanHandle*(), that captures the notion of whether a given handler can, in its current state, handle a given request.

The role contract for Handler identifies _succ as the only required role field. This field will store a reference to the next handler in the chain, or will be set to null if no such handler exists. The only named method of the role is handle(), which receives a request as argument and returns a response. The pre-condition of this method requires that if the current handler is not able to handle the request, as determined by the *CanHandle*() concept, there must be a successor to which the request can be forwarded. Interestingly, the standard informal description of the pattern does *not* impose such a condition. Instead, "the request can fall off the end of the chain without ever being handled" [9]. We feel that most designers will not want such behavior, and have hence included the requirement of a non-null successor if a request cannot be handled[12]. The **ensures** clause of handle() asserts that if the request r can be handled by the current handler, the trace of the method will be empty. Otherwise, there will be one call to the handle() method of the object referenced by the _succ field, with the same request supplied as argument. The result returned by that call will be returned as the result of the original.

Let us now turn to the the behavior that the use of the *CoR* pattern ensures. Unlike the Observer pattern, the intent of the *CoR* pattern isn't to ensure that a particular state assertion is satisfied. Instead, the pattern guarantees that when a handle() request is received by h, one of the enrolled objects, h will handle the request if it is capable of doing so. Otherwise, the request will be forwarded to h_n, another object that is capable of handling the request, and is reachable from h in the chain. Further, the value returned by h_n will be returned by h as the result of the original call to h.handle(). This is captured by the *invariant result* specified in our contract. Note that this assertion also specifies that none of the intermediate objects in the chain (between h and h_n) are

[12]Note that our contract does not guarantee that the request will ultimately be handled since it is possible for the handler chain to have cycles. This again could be addressed by adding appropriate clauses to *RRI*.

capable of handling the request. Indeed, this is the reason the request was ultimately forwarded to h_n.

Let us now briefly consider the subcontract corresponding to the *CoR* specialization applied in the extended SimSickCity system. The Nurse class plays the Handler role, with handle() being mapped to the pStatus() method. Patient plays the Request role, and String plays the Response role. The *CanHandle*() concept is defined as follows[13]:

CanHandle(Nurse n, Patient p)
  { return(n.pInfo.keySet().contains(p)); }

This definition simply states that a nurse n can handle a request for status information about a patient p provided n is currently watching p. Given this definition, it is straightforward to see that the pStatus() method meets the requirements specified in Fig. 15 for Handler.handle(). Therefore, by appealing to the invariant result that the use of the pattern ensures, we can conclude that our system exhibits the result we previously claimed, thereby completing our reasoning task.

There is one final point concerning our *CoR* contract that is worth mentioning. Standard informal descriptions of the pattern suggest that requests may be organized in an inheritance hierarchy, with different handlers in the chain responsible for handling requests of each type. Our approach handles such cases by allowing designers to provide suitable definitions for the *CanHandle*() concept. In such cases, the definition of the concept will be based on the particular class that the given request is an instance of, and whether the particular handler is the one corresponding to that class. But, as in the case of the extended SimSickCity system, *CanHandle*() may also depend on more detailed (and dynamic) information concerning the request and the handler. Thus, as in the case of the *Observer* pattern, the process of formalizing *CoR* in our approach enabled us to identify dimensions of flexibility that were not evident in the informal description of the pattern.

4 Experiences with Other Patterns

We have applied our specification and reasoning approach to a number of other common patterns, and have made the resulting pattern contracts available for download from our web site [33]. In this section, we briefly summarize our experiences specifying three of these patterns: *Iterator*, *Memento*, and *Composite* [9].

[13]Since all the nurse objects are arranged in a single chain, there is only one instance of the pattern in this scenario. Hence, concerns regarding the lead object of the pattern instance are not as important as in section 3.3. If, however, the nurse objects were arranged in multiple chains, these concerns would again become important.

Iterator: The intent of the Iterator pattern is to allow a client object to iterate through the elements contained within an aggregate object without exposing the aggregate's storage representation. This is achieved by delegating the element traversal strategy to an iterator object created by one of the aggregate's methods. The iterator encapsulates references into the aggregate, and provides an interface for traversing the elements one at a time. It is possible for each type of iterator to support a different traversal strategy[14].

Our Iterator contract is based, in part, on the component specifications presented in [37]. The pattern contract includes three role contracts: Aggregate, Iterator, and Data. The first two roles correspond to pattern participants. The third role serves as a *type parameter* used to capture the types of objects stored within the aggregate. The Aggregate role contract defines the state component _elements, corresponding to the Multiset of objects stored by the aggregate[15]. The Iterator role contract defines _remaining and _traversed, corresponding to the Multiset of objects remaining in the traversal and those that have already been traversed, respectively. The role contract additionally defines _current, corresponding to the current object in the traversal represented by the iterator.

One important aspect of the Iterator pattern is that an iterator need not traverse every element within the aggregate; instead, only those items in the aggregate that pass a specified *test* are included. To allow for this, our pattern contract defines the auxiliary concept *IncludeInTraversal*(i, d), used to filter the elements included in a traversal. The relation is defined on an iterator object and a data object, and evaluates to true if and only if d should be included in the traversal represented by i. The ensures clause of Aggregate.createIterator() requires (*i*) that a new iterator be returned, (*ii*) that the object's _remaining field contain the elements selected for traversal from the aggregate's _element field (as filtered by *IncludeInTraversal*()), (*iii*) that the object's _current field be equal to null, and (*iv*) that the object's _traversed field be empty. The Iterator role contract specifies methods for controlling the behavior of the iterator, and for accessing the _current field. The most

[14]The original pattern description identifies two types of iterators: *external* and *internal*. An *external* iterator exposes an iteration interface used to traverse the elements contained within an aggregate. An *internal* iterator does not expose an iteration interface, and provides a single method for applying a particular function, passed as argument, to the elements contained within an aggregate. In our discussion here, we focus only on *external* iterators, which, based on iterators that appear in various commercial class libraries, is, by far, the more commonly used kind of iterator.

[15]We use Multiset in the pattern contract, rather than Set, since aggregate objects may contain duplicate values.

interesting method is Iterator.next(), which removes the *next* element from _remaining, adds the element to _traversed, and sets the _current field equal to the element. But how does the method determine the *next* element? This information is captured by the auxiliary concept *CanAppearBefore(i, d1, d2)*, a relation defined on an iterator object and two data objects. The relation evaluates to true if and only if d1 can appear *before* d2 in the traversal represented by i. When applied to the elements in _remaining, the resulting ordering makes it possible to specify the Multiset of objects suitable to serve as the *next* element in the traversal.

The pattern invariant for the Iterator contract is included below, and is worth considering briefly.

$$
\begin{aligned}
&\mathsf{Aggregate}(\mathsf{players}[0]) \land \\
&\mathsf{Iterator}(\mathsf{players}[1:]) \land \\
&(\forall i : (\mathsf{i} \in \mathsf{players}[1:].\mathsf{objs})) :: \\
&\quad (\mathsf{players.objv}[0]._\mathsf{elements}^{\otimes IIT} = \\
&\qquad (\mathsf{i}._\mathsf{traversed} \cup \{\mathsf{i}._\mathsf{current}\} \cup \mathsf{i}._\mathsf{remaining})) \land \\
&\quad (\forall d1, d2 : (\mathsf{d1} \in (\mathsf{i}._\mathsf{traversed} \cup \{\mathsf{i}._\mathsf{current}\}) \land \\
&\qquad\qquad (\mathsf{d2} \in \mathsf{i}._\mathsf{remaining}))) :: \\
&\qquad \mathit{CanAppearBefore}(\mathsf{i}, \mathsf{d1}, \mathsf{d2})
\end{aligned}
$$

The first clause of the invariant states that the first object to enroll in an instance of the pattern, and therefore the lead object for that instance, will be an aggregate object. The second clause states that all other enrolled objects will play the role of Iterator. The next clause relates the states of the iterator objects and the state of the aggregate. This clause specifies that the elements traversed by the iterator, combined with the current element in the traversal, and those that are yet to be traversed, are the same elements contained within the aggregate, less those filtered out by the *IncludeInTraversal*() concept. (We use the superscripted $\otimes IIT$ notation to denote the multiset obtained by projecting out the data objects for which *IncludeInTraversal*(i, d) evaluates to true.) The final clause of the invariant relates the elements in the traversal represented by each iterator i. This clause specifies that each of the elements that have already been traversed, plus the current element in the traversal, will be allowed, according to the definition of *CanAppearBefore*(), to appear *before* the elements that are yet to be traversed. Stated another way, this clause specifies that each iterator object will traverse the aggregate elements in the desired order.

An interesting issue that we encountered while developing this contract is that certain applications of the pattern require that the aggregate remain unchanged while its iterators are in use. Hence, whenever an aggregate object is altered, this variation of the pattern requires that the corresponding iterators created before that point be marked as *defunct*. Defunct iterators may not be used again. To accommodate this possibility, we introduce a simple auxiliary concept, *AllowAggregateModifications()*, that allows subcontract designers to specify whether modifications to the aggregate are allowed. We then impose suitable conditions on the *global* call sequence, which, if aggregate modifications are *not* allowed, prevent accessing the methods of *defunct* iterator objects.

Memento: The intent of the Memento pattern is to allow an object to take a *snapshot* of an originator object's state without violating encapsulation, so that the snapshot can, if necessary, be used to restore the state of the originator at some later point. The *snapshot* created by the originator is a memento object. This pattern is often used when implementing *undo* operations, such as those found in common graph editors.

The Memento pattern contract consists of two role contracts: Memento and Originator[16]. The Originator role contract specifies the methods createMemento() and setMemento(), for creating and applying memento objects, respectively. The Memento role contract does not specify any methods, since the precise manner in which the memento's state is accessed is not essential to the pattern, and will vary from application to application. Perhaps more surprising is that neither of these roles define any state components. Instead, the behaviors required of the named and unnamed methods are expressed using auxiliary concepts. The *MemCopy*(o, m) concept, for example, is a relation defined over the state of an originator and the state of a memento, and is used to capture whether the state m represents a *valid snapshot* of the state o. This concept is used in the specification of createMemento() to require that the newly-created memento object contain appropriate information about the originator. The *Restored*(o1, o2) concept is a relation defined on two originator states, and captures whether o2 is similar enough to o1 to be a *proper restoration* of o1. This concept is used in the specification of setMemento(m) to require that the originator be restored to a state that is *similar enough* to the state that it was in when m was created.

The final concept included in the pattern contract is *SameMem*(m1, m2), a relation defined over two memento states. This concept corresponds to the notion of whether the originator information contained in the state m1 is the *same essential* originator information contained in the state m2. This concept is used to specify the behavior required of unnamed Memento methods. In effect, unnamed methods are required to leave the memento in a state that contains the same

[16]The original pattern description identifies the Caretaker role. Since, however, there are no implementation responsibilities or behavioral guarantees associated with this role, we omit it from our contract.

essential information about the originator as the state it was in when the method was invoked. Note that if *SameMem*(m1, *m2*) evaluates to *true*, it doesn't necessarily mean that m1 and m2 are identical. The definition of *SameMem*() varies from one application to another. In the design of a *graph editor*, for example, it might be the case that the nodes and their connectivity are important, but their positions are not. In this case, we would provide a definition of *SameMem*() that allows unnamed Memento methods to affect the layout of the graph, but not the nodes or their connectivity. This scenario again illustrates the additional dimensions of flexibility identified in our pattern contracts. Whereas standard informal descriptions of the pattern suggest that memento objects are immutable, our formalization allows modifications that do not compromise the intent of the pattern.

Composite: The Composite pattern is a *structural pattern*, whose intent is to express a series of whole-part relationships in a tree structure that allows composite objects to be treated in the same manner as leaf objects. Both types of objects, Composite and Leaf, share a common interface inherited from Component. When a method is invoked on a composite object, the object performs the appropriate actions, and then invokes the same method on each of its *children* — the component objects it references.

The Composite pattern contract defines three role contracts: Component, Leaf, and Composite. To capture the inheritance relations required by the pattern, Leaf and Composite are required to inherit from Component. As a consequence, in a subcontract, classes that map to either Leaf or Composite must inherit from a class mapped to Component. Somewhat surprisingly, there are no auxiliary concepts defined, and only the Composite role contract defines state. The _children field maintains a Set of component objects corresponding to the composite's children in the tree. The Composite role contract specifies methods for adding and removing components from _children. Both Leaf and Composite define Operation(), a method inherited from Component, which corresponds to the method that the client is interested in performing on both kinds of objects. For a leaf object, how this operation is performed will of course depend on the particular type of primitive component it is. Hence, the contract does not impose any conditions on the behaviors that must be exhibited by this method. The specification for the Composite role, however, requires that the method invoke Operation() on each of the component objects in _children, with the same argument values passed to the original call. The specification does not, however, impose any additional conditions on the behavior of the operation. Given the structural nature of this pattern, specifying such conditions would be overly-restrictive.

It is worth mentioning that when the Composite pattern is applied in practice, there may be several methods defined in Component that should be implemented by objects playing the Leaf and Composite roles. To accommodate such scenarios without requiring one subcontract for each Component method, our contract allows multiple methods to be mapped to Operation(). This presents a problem, however, since the specification of Composite.Operation() requires the same *actual* method to be invoked on each of the composite's children — not *any* method mapped to Operation(). Hence, we introduce the playerMethod notation, which allows us to refer in a call sequence condition, to the *actual* player method in question. This notation is somewhat analogous to the use of the this and lead keywords, and allows us to specify in the ensures clause of Composite.Operation(), that implementations of the method must invoke the corresponding *actual* method on each of their children — not *some* method mapped to Operation().

5 Related Work

Much of the design patterns literature focuses on documenting patterns informally, typically in a style similar to the one introduced by Gamma *et al.* [9]. Some authors have also considered the question of how to informally document the ways in which patterns are applied in particular systems. Vlissides [35], for example, proposes extensions to the Unified Modelling Language (UML) [28] based on Venn diagrams. His extensions identify the patterns used in a system, the classes involved in each of the pattern implementations, and the roles associated with each class. Schauer and Keller [29] describe a reverse engineering tool that generates similar diagrams. Dong [5] proposes additional extensions that describe how the methods and attributes of a class map to the methods and attributes required to implement its role behaviors. In contrast to these approaches, our goal has been to develop a formalism in which patterns and their applications can be specified and reasoned about *formally*. Other authors have considered similar issues. In this section, we consider the approaches that are most directly relevant to our work.

Eden *et al.* [6, 7, 8] describe an approach in which design patterns are represented as formulae within a higher-order logic formalism. 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. The formalism enables the specification of rich structural properties, but

provides only limited support for behavioral properties. In the specification of the Observer pattern, for example, there is no characterization of the precise conditions under which Notify() must be called, nor of the conditions that must be satisfied by the Update() method. By contrast, behavioral properties play a central role in our work — but not at the cost of excluding structural properties. The role contracts presented as part of the Observer and CoR contracts, for example, impose requirements on the fields that must be supplied by objects playing these roles. The Composite contract captures more complex structural properties, such as the inheritance relations among the participating objects. At the same time, all of these contracts capture important behavioral properties. Indeed, as we have seen, even patterns that focus on structural properties have important behavioral components that must be satisfied.

Mikkonen [24] describes an approach in which patterns are specified using *DisCo* [22, 18, 17], a specification language for reactive systems that uses an action system model similar to UNITY [4]. Data classes model pattern participants, and guarded actions model their interactions. The approach focuses primarily on behavioral properties. For example, it allows us to specify the order in which role methods must be invoked. The specification of the Observer pattern, for instance, states that Update() must be invoked on each attached observer following a call to Notify(). Mikkonen also introduces a notion of *refinement* that allows us to specify the classes and class methods corresponding to the roles and role methods in a particular pattern application. However, the specifications seem to omit important properties. The Observer specification, for example, does not require that Notify() be provided by the subject object; it may be provided by *any* object. Further, the specifications are overly-constrained since there is nothing analogous to our use of auxiliary concepts in the formalism. Thus, the Observer specification requires that each observer object contain *exactly* the same state as the subject object to which it is attached. Upon completion of the Update() method, the observer state is required to be identical to that of the subject. Similarly, the specification of Notify() requires this method to be invoked whenever there is *any* change in the subject's state. These limitations mean that pattern *refinement* is essentially a mapping between names, and does not consider behavioral considerations. Despite these limitations, however, the use of an action-logic notation is useful since it allows us to express complex call sequence requirements in a natural manner. In our future work, we will consider adopting similar notations to express conditions on call sequences.

Helm *et al.* [12, 16] describe a *contract* formalism that shares similarities with our work. In their approach, each contract specifies pattern participants in a manner similar to our use of role contracts, and a state invariant that captures the intent of the pattern, similar to our use of pattern invariants. The formalism also supports contract specialization, including the ability to tailor the definitions of certain relations, similar to our use of auxiliary concepts. There are, however, a number of important differences in our work. The formalism described by Helm *et al.* does not, for example, provide a means of imposing conditions on the definitions that may be supplied for the various relations used within a contract. As we have seen, such conditions are required to prevent incompatibilities between definitions, which can ultimately lead to the pattern intent being violated. In addition, their formalism does not provide a construct analogous to the *others* clause used in our role contracts. As a consequence, unnamed methods provided in a pattern specialization can perform arbitrary actions, thus again violating the intent of the pattern. Further, although the formalism allows us to specify that certain methods be invoked at particular times to achieve certain conditions, there is nothing to prevent additional methods from being invoked that then nullify that condition. In the Observer pattern contract, for example, there is nothing to prevent the Notify() method from making an arbitrary change to the subject *after* having invoked Update() on each attached observer. Our use of call sequences prevents such problems. It is important to mention, however, that one important idea in their work is that contracts may be composed to construct "higher-level" contracts. A similar idea should be applicable to our specifications, and we plan to investigate this possibility in future work.

To an extent, our work is also related to the work on *architecture description languages* (ADLs), such as *Rapide* [23], *UniCon* [31] and *Wright* [10]. One important idea used in both *UniCon* and *Wright* is the notion of a component *connector*. While *components* in these formalisms correspond to roles in our pattern specifications, *connectors* represent the interaction protocol between two or more components. For example, a connector in *Wright* specifies the names of the roles that participate in the interaction pattern represented by the connector, and specifies the allowed patterns of interactions using a *CSP*-like [14] notation. *Rapide* is also concerned with interaction patterns between various components of a system, but does not use a connector-like construct. Instead, the interaction patterns between the *modules* under consideration are specified in terms of *posets* (partially ordered sets) of events. The use of posets makes it possible to compactly represent a number of different possible event orderings. Although concurrency-related

considerations were the main motivation for the use of posets in *Rapide*, posets may also be of use in specifying method calls required by a pattern under given conditions. For example, using posets, we can conveniently state that the Subject.Notify() method is required to call Update() on each attached observer without constraining the order of these calls. More important, we believe that some of the ideas in ADLs, including such notions as connectors, will be of considerable help in applying our approach to specifying *architectural* patterns [3, 30].

Heuzeroth *et al.* [13] describe a static analysis tool that identifies the patterns used in a system based on a set of input pattern specifications. The authors use a version of Prolog as a specification language. This choice simplifies the design of the tool since it can leverage the Prolog interpreter to evaluate potential matches between classes, class methods, etc., to pattern roles, role methods, etc. But specifications of this kind are unwieldy. As an alternative, the authors introduce *SanD*, a language that offers OO constructs, and provides primitives to specify call sequence conditions such as those imposed on Subject.Notify() in the Observer pattern contract. Although specifications developed using this notation are more readable than their Prolog counterparts, they still appear to be designed for use by the tool, rather than by designers. More important, the specifications are overly-constrained since there is nothing analogous to auxiliary concepts, general role maps, etc. Nevertheless, the idea of a reverse engineering tool that can identify design patterns is an interesting one. We believe, however, that while such tools are of value in dealing with legacy code, when designing new systems, a more useful tool would be one that, given information about the patterns used in a particular system, can monitor the system at runtime to see if the appropriate pattern requirements are respected. We will consider this point further in the next section.

Hanneman and Kiczales [11] take a different approach to capturing patterns. Rather than providing reusable *specifications* that can be specialized to particular applications, they provide reusable pattern *implementations* that can be specialized. Their approach is based on the use of *AspectJ* [20]. The aspects corresponding to a given pattern implementation encapsulate the role interactions required of *any* specialization of the pattern. These aspects are declared as *abstract*, and defer the portions of the implementation that vary to one or more subaspects. In this sense, the abstract aspects are analogous to our contracts, and the concrete subaspects are analogous to our subcontracts. Consider, for example, the Hanneman and Kiczales implementation of the Observer pattern. The abstract aspect maintains a mapping between observer objects and the subject objects to which they are attached, and provides methods for attaching and detaching observers. The *after advice* bound to the *pointcut* that captures Subject method invocations that modify the state of the subject consists of a sequence of calls to the Update() method of each attached observer. Definitions of this pointcut are provided in the subaspects corresponding to particular applications of the pattern. Hence, the notion of *modification* can be specialized.

We should emphasize, however, that this is a pattern *implementation*, not a *specification*. Hence, a natural question to consider is whether the implementation satisfies the requirements specified in our Observer pattern contract. To answer this, we analyzed the aspect implementation, focusing on the methods and advices. Interestingly, it turned out that the implementation of the Observer pattern in [11] violates a subtle requirement: When a new object attaches as an observer to a given subject, the implementation in [11] adds the attaching object to the set of objects observing the given subject, but does not invoke Update() on the object (as required by our contract). Hence, between the point of attachment and the point at which the subject's state is next modified, the attached observer may be in a state that is *inconsistent* with the subject. Thus, although our contract for Observer was designed with standard class-based implementations in mind, it turned out to be useful for checking the correctness of an aspect-based implementation, as well.

Before concluding this section, it is worth mentioning that key aspects of our approach are related to important issues identified by some of the authors who use an informal approach to document patterns. Buschmann *et al.* [3], for example, in summarizing how patterns should be used in building practical systems, state, "You should be able to reuse the pattern in many implementations, but so that *its essence is still retained* [emphasis added]. ... After applying a pattern, an architecture should include a particular structure that provides for the roles specified by the pattern, but *adjusted and tailored to the specific needs of the problem at hand* [emphasis added]." What is the *essence* of a pattern and how do we ensure that it is retained? The answer, provided by our work, is that the pattern contract captures its essence — specifically, the specifications of the *named* methods and the *other* methods of each role, the constraints on the auxiliary concepts, and the pattern invariant. Similarly, the subcontract corresponding to a given application of the pattern tells us exactly how the pattern has been adjusted and tailored to the needs of the particular problem — specifically, the *state map*s and *method map*s corresponding to each class playing a given role, and the def-

initions of the auxiliary concepts of the pattern. Thus, our approach provides clear and precise ways to formalize the key aspects of design patterns and their applications, and serves as a natural complement to informal approaches to patterns.

6 Discussion

The contributions of our work have been two-fold. First, we developed an approach to precisely specifying the requirements that must be satisfied when applying a given design pattern, as well as the behavioral guarantees that accrue as a result. Second, we developed a reasoning approach that can be used to show that a pattern's implementation requirements have indeed been satisfied, and to arrive at the system behaviors that should be expected as a result. While achieving this precision has been our primary goal, another important goal was to ensure that pattern flexibility is not compromised as a result of formalization. We achieved these seemingly conflicting goals by parameterizing our specifications in terms of *auxiliary concepts* and *role maps*. These structures allow us to define *pattern contracts* that capture the properties common across all specializations of a particular pattern, and *pattern subcontracts* that capture the properties specific to particular pattern specializations. We demonstrated in our case study how system designers can use these formalisms to reason about their systems in a *pattern-centric* fashion, rather than in the standard *class-centric* fashion, to arrive at the system properties that will be exhibited as a result of the patterns used in their design.

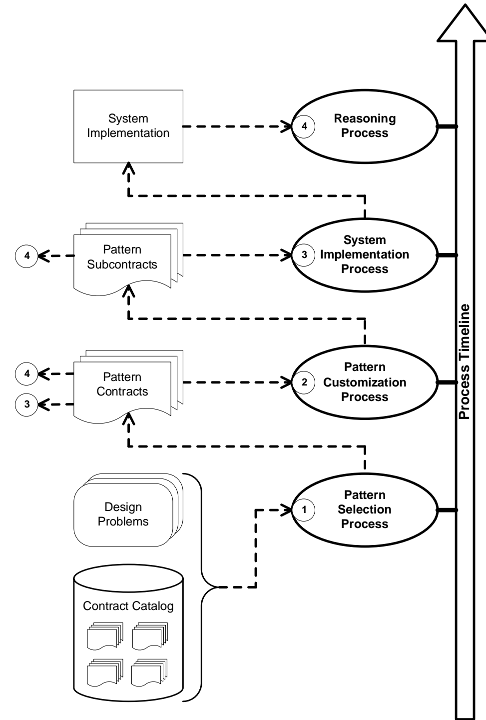

Fig. 16. Software Development Process

Fig. 16 illustrates the pattern-centric software development process enabled by our approach. Note that the figure omits some of the standard phases in the software lifecycle, such as requirements analysis, testing, etc. In the first step, *the pattern selection process*, a catalog of pattern contracts for the most commonly applied patterns, along with their usual informal descriptions, is assumed to be available. Based on the contracts and the informal descriptions, as well as the requirements of the system in question, the design team identifies the patterns that will be used in the system's design. In the next step, the *pattern customization process*, the team determines how the patterns will be customized for use in the system. Based on these customization decisions, the team then develops the subcontracts that characterize the ways in which the patterns will be specialized. Next, the *system implementation process* proceeds. This step is guided by the requirements specified in the relevant pattern contracts, as specialized by the mappings and concept definitions defined in the corresponding subcontracts. In practice, we can expect these two steps to be iterated numerous times, since as the implementation process proceeds, it may be necessary to redefine parts of the subcontracts and to appropriately modify the implementation. Finally, once the system is partially or completely constructed, the *reasoning process* proceeds, which includes the types of activities we sketched for the SimSickCity case study. These reasoning activities are based on the implementation code, as well as the relevant contracts and subcontracts, and help to ensure that the system meets the requirements of the pattern, and to check that the expected system behaviors match the requirements for the system.

In practice, software projects do not typically go through formal reasoning phases. So one could reasonably ask, how realistic is our proposed development process? Note first that the creation of the pattern contract catalog is, like the creation of informal pattern descriptions, a one-time effort for the community. Individual designers and design teams must read and understand the contracts, but they are not responsible for developing them. The main reasoning activities for designers consist of defining the subcontracts corresponding to the pattern specializations used in their designs (step ②), and applying the reasoning process used to show that their systems meet the requirements specified in the relevant pattern contracts, specialized appropriately (step ④). Defining the subcontracts is normally a straightforward task since the subcontracts only capture infor-

mation about how patterns are applied in particular systems. Indeed, as we saw in Section 2, portions of each subcontract are written in the form of code, making the subcontract language immediately familiar to most designers.

The reasoning task in step ④ poses a greater difficulty for developers who are not trained in formal methods. This can be considerably mitigated by using *runtime monitoring* in place of (or in addition to) formal reasoning. We are currently developing a tool that will automatically generate, given the contracts for the patterns used in a system and the subcontracts corresponding to their specializations, runtime monitoring code that can be executed with the system code to detect pattern implementation errors. The monitoring code will check, at appropriate points during the system's execution, that the relevant requirements are satisfied. In the current version of the tool, we assume that the system is implemented in *Java*. The monitoring code generated by the tool is in *AspectJ* [21]. An *aspect-oriented* approach was a natural choice since many of the pattern contracts are concerned with multiple classes (or more precisely, roles), and hence involve *cross-cutting concerns*. This approach is similar to runtime monitoring approaches that check method pre-conditions, method post-conditions, and class invariants. The main difference is that because pattern contracts involve multiple classes, for example the invariant in the Observer pattern contract (Fig. 5), the assertions that must be checked cut across the system hierarchy. A preliminary version of our monitoring approach is described in [34]. When the monitoring code produced by our tool detects a runtime violation of the requirements specified in the pattern contracts, it produces a suitable warning message. This feedback helps developers locate the points where their system does not abide by the pattern requirements, and may therefore indicate potential problems. In practice, we expect designers to do some of the reasoning called for in step ④ and to fix any problems revealed through that process, and then to use the monitoring code to check for additional defects. Thus, step ④ is perhaps better labelled as *reasoning and monitoring process*.

As we noted earlier, the development process illustrated in Fig. 16 omits some of the standard steps in the software lifecycle. One key step, for example, is that of *maintenance and evolution*. The pattern-centric approach that we have proposed provides additional benefits in this phase. One benefit is that the subcontracts corresponding to the various specializations of the patterns used in a system are a valuable part of the system's *documentation*. The software engineers responsible for maintenance and evolution are able to consult this documentation in determining whether potential changes would be consistent with the underlying design philosophy of the system. We saw a simple example of this in Section 3.3 where we considered a possible change to the Patient class of the SimSickCity system. As we saw, the change we considered, although reasonable from the point of view of the Patient class, would violate the relation implied by the use of the Observer pattern, between the Nurse class and the Patient class. This would manifest itself as a violation of a contract requirement. Our arguments in Section 3.3 were based on reasoning about the system, but the same violation would also be discovered using a monitoring tool. Thus our approach will enable the maintenance team to ensure that any changes made to the system do not violate the design integrity of the system.

We now turn to possible directions for future work. One important area is the development of a *catalog* of contracts for the most commonly applied patterns. As we noted in Section 4, we have developed a number of additional contracts, in addition to those considered in Section 3. We are currently working on several others, including Bridge, Builder, Decorator, Facade, Mediator, and Visitor. When the contracts for these and other patterns are complete, we will publish them on the web for use by the software engineering community. The site will also include small and medium-sized case studies of systems built using the patterns. The case studies will provide details of the subcontracts corresponding to the pattern applications, and details about the behavioral results that follow.

A second area of future work considers the *adequacy* of our formalism. There are, in fact, two parts to this question. The first part concerns the adequacy of the language for expressing different types of constraints and invariants that might be appropriate to various patterns. Consider again the *Chain of Responsibility*. In the (partial) contract for this pattern in Section 3, the invariant specifies that if a given handler object h returns a result re in response to a handle() request that it receives, it must be the same result returned by an appropriate handler in the chain beginning at h. While this is accurate, it doesn't quite state that h will necessarily pass it on to its successor if it is not able to handle the request, nor that this process will repeat at each handler. It only states that if h returns a result re, then this must be the same result returned by the first object in the chain beginning at h that can handle the request. But a more natural statement of the required behavior would be that if h receives a request r, this request will lead to a sequence of calls along an appropriate chain of objects in which r is forwarded from one handler to the next. Expressing this would require us to use suitable *temporal* operators or,

as noted in Section 5, *action logic* notations. The disadvantage is that such specifications can be somewhat more difficult to comprehend and reason about than first order logic assertions. Nevertheless, for certain patterns, the added expressivity would be valuable.

The second part of the adequacy question concerns the efficacy of our approach in dealing with different types of patterns. To this point we have considered patterns intended primarily for use in sequential systems. While many of these patterns are generally applicable, there are also patterns for handling issues specific to particular system classes. Patterns for networked and concurrent systems, for example, address issues involving synchronization, concurrency, etc. Consider, for example, the *Active Object* pattern [30]. The intent of the pattern is to improve concurrency while simplifying synchronization by transforming system objects into threaded servers (or *actors* [1]). In effect, the pattern decouples the invocation of a method from its execution. A key requirement of the pattern is to perform the decoupling in a way that preserves the client's view of the invocation as being *normal*. For us to capture this requirement in the corresponding contract, we would have to extend our formalism to include suitable ways of referring to processes, threads of control, etc. Similarly, when considering patterns that dictate specific communications between processes, the formalism would have to provide ways to refer to the processes playing particular roles, the sequences of inter-process communications, etc. Further, we would have to consider the possibility that the auxiliary concepts in such patterns might depend on these communications. In our future work, we will provide suitable extensions to the formalism to deal with these possibilities. One interesting point to note is that a natural way to deal with communications between processes in a distributed system is via *communication traces* [15], which are similar to the traces we have used for recording information about sequences of method calls. Another way to deal with the sequences of communications would, of course, be via action-logic specifications. We intend to explore both approaches in our future work.

Another line of future work concerns the development of tools that help application developers achieve the maximum benefit of our approach when dealing with practical systems. As noted earlier, we have built a prototype system that automates the generation of runtime monitoring code based on the pattern contracts and subcontracts used in a system's design. The generated code can be used to detect pattern contract violations as early in the development process as possible. While this tool is useful during pattern-centric debugging and testing, it is of less value for someone — say, a new member of the design team — who wishes to explore the behavior of the system with the intent of developing a pattern-centric understanding of its structure and behavior. To help address this need, we intend to develop a visualization tool that will present a dynamic, pattern-centric view of the system's execution. During execution, information will be recorded about the various pattern instances that are created, objects that enroll in these instances, method invocations on these objects, invocations made from within these objects, etc. When the system terminates, the visualization tool will allow us to 'play-back' the execution, presenting a pattern-centric view of the recorded behavior. The tool will allow the user to focus on particular patterns, particular instances of those patterns, particular objects enrolled in those instances, particular method invocations, etc. We believe that such a tool will be of significant benefit, especially in the development of large systems with correspondingly large teams.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MITP, 1986.

[2] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings of the Eighth ECOOP*, pages 139–149, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: A system of patterns*. Wiley, 1996.

[4] K.M. Chandy. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[5] J. Dong. UML extensions for design pattern compositions. *Journal of Object Technology*, 1(5):151–163, November / December 2002.

[6] A.H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Tel Aviv University, 2000.

[7] A.H. Eden. Formal specification of object-oriented design. In *Proceedings of the International Conference on Multidisciplinary Design in Engineering (CSME-MDE '01)*, November 2001.

[8] A.H. Eden. LePUS: a visual formalism for object-oriented architectures. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT '02)*, June 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[10] D. Garlan and R. Allen. Formalizing architectural connection. In *Proc. of 16th ICSE*, pages 71–80. IEEE, 1994.

[11] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In C. Norris and J.B. Fenwick Jr., editors, *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, ACM SIGPLAN Notices, pages 161–173, New York, November 2002. ACM Press.

[12] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*, pages 169–180. ACM Press, 1990.

[13] D. Heuzeroth, S. Mandel, and W. Lowe. Generating design pattern detectors from pattern specifications. In *Proc. of the 18th Int. Conf. on Automated Softw. Eng.* IEEE, 2003.

[14] C.A.R. Hoare. Communicating sequential processes. *Comm. ACM*, 21:666–677, 1978.

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] I.M. Holland. Specifying reusable components using contracts. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP '92)*, volume 615, pages 287–308. Springer-Verlag, 1992.

[17] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: Marriage of actions and objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS '91)*, pages 142–151. IEEE Computer Society, 1991.

[18] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering (ICSE '90)*, pages 63–71. IEEE Computer Society, 1990.

[19] R. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proc. 15th ECOOP*, pages 327–353. Springer-Verlag, 2001.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, pages 327–353, London, UK, 2001. Springer-Verlag.

[22] R. Kurki-Suonio and H.-M. Järvinen. Action system approach to the specification and design of distributed systems. In *Proceedings of the 5th International Workshop on Software Specification and Design*, number 14 in ACM Software Engineering Notes, pages 34–40, 1989.

[23] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21:336–355, 1995.

[24] T. Mikkonen. Formalizing design patterns. In *Proceedings of the 20thth International Conference on Software Engineering (ICSE '98)*, pages 115–124, Washington, DC, 1998. IEEE.

[25] T. Reenskaug. *Working with objects*. Prentice-Hall, 1996.

[26] D. Riehle. Composite design patterns. In *Proc. of OOPSLA*, pages 218–228. ACM, 1997.

[27] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

[28] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[29] R. Schauer and R. Keller. Pattern visualization for software comprehension. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, pages 4–12, Washington, DC, 1998. IEEE Computer Society.

[30] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture: Patterns for concurrent and networked objects*. Wiley, 1996.

[31] M. Shaw, R. Deline, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21:314–335, 1995.

[32] N. Soundarajan and J.O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In

A. Finkelstein, J. Estublier, and D. Rosenblum, editors, *Proc. of 26th Int. Conf. on Software Engineering (ICSE)*, pages 666–675. IEEE Computer Society, 2004.

[33] N. Soundarajan, J.O. Hallstrom, and B. Tyler. Specifying, monitoring, and visualizing design patterns. http://www.cs.clemson.edu/ jasonoh/dsrg/smavdp.

[34] N. Soundarajan, J.O. Hallstrom, and B. Tyler. Specifying and monitoring design pattern contracts. In *Proc. of the SAVCBS 2004 Workshop (ICSE)*, pages 87–94. www.cs.iastate.edu/SAVCBS, 2004.

[35] J. Vlissides. Notation, notation, notation. *C++ Report*, April 1998.

[36] J. Warmer and A. Kleppe. *The Object Constraint Langauge*. Addison-Wesley, 1999.

[37] B.W. Weide, S.H. Edwards, D.E. Harms, and D.A. Lamb. Design and specification of iterators using the swapping paradigm. *IEEE Transactions on Software Engineering*, 20(8):631–643, 1994.