

# Tracing Correct Usage of Design Patterns

Neelam Soundarajan<sup>1</sup>, Johan Dovland<sup>2</sup>, Jason O. Hallstrom<sup>3</sup>

<sup>1</sup> Computer Sc. & Engineering  
Ohio State University  
Columbus, OH 43210  
USA  
neelam@cse.ohio-state.edu

<sup>2</sup> Dept. of Informatics  
University of Oslo  
PO Box 1080 Blindern  
N-0316 Oslo, Norway  
johand@ifi.uio.no

<sup>3</sup> Computer Science Dept.  
Clemson University  
Clemson, SC 29634  
USA  
jasonoh@cs.clemson.edu

## Abstract

*We previously described a contract formalism for specifying design patterns precisely, and showed how the formalism amplifies the benefits of pattern usage. In this paper, we present an extension to the formalism that addresses problems which arise in systems involving complex interconnections among objects, including potential cyclic reference structures. In the presence of such structures, the behavior of an apparently simple operation can be quite complex. In this paper, we develop an approach that accounts for these complexities by formalizing pattern behaviors in terms of pattern-instance traces, ghost variables that record method invocations and completions on the objects participating in a pattern instance. We illustrate the approach by considering the standard Observer pattern.*

## 1 Introduction

Design patterns [8, 3, 14] have had a profound impact on design practice, especially in the context of object-oriented design. In previous work [16, 9, 17], we argued that in order to maximize the benefits of patterns, we must have precise specifications of patterns and the ways in which they are applied in particular systems. We presented an approach to providing such specifications in the form of *pattern contracts* and *subcontracts*. A pattern contract captures the requirements and behavioral guarantees that apply to *all* uses of a particular pattern, while a subcontract characterizes how the pattern is specialized for use in a *particular* application.

In this paper, we consider issues that arise when reasoning about the use of patterns in *large* systems, systems which may involve complex interconnections among the participating objects, including cyclic reference structures. In the presence of such structures, the net behavior of seemingly simple method calls can be surprisingly complex. Suppose, for example, a method  $m()$  is invoked on an object  $o$ , and that during its execution  $m()$  invokes a method  $n()$  on an object  $o'$ . Due to the in-

terconnections among  $o$ ,  $o'$ , and other system objects, the invocation may affect changes not just in the state of  $o'$ , but of other objects in the system, *including*  $o$ . Indeed, it is possible that before the invocation of  $n()$  completes, the method  $m()$  is itself invoked again on  $o$ . Such a situation may seem contrived but this precise scenario arises naturally in typical pattern applications, as we will see in a simple example later in the paper. Neither the pattern contracts of the kind presented in our previous work [16, 9, 17], nor those considered by other authors [6, 7, 12] account for such situations. In effect, implementations of this sort are disallowed by these contracts. Our goal in this paper is to develop a specification framework that does allow for such implementations.

In the approach presented here, the concept of a *pattern instance trace*, or *pi-trace*, plays a central role. A pi-trace is a record of invocations and completions of method calls on all of the objects participating in a *pattern instance*. This may be contrasted with an *object trace*, which records method calls placed on a *single* object. The latter structure aligns more naturally with the most common approach to reasoning about OO systems — reasoning in terms of the constituent system objects and the methods invoked on them. While object-level reasoning is valuable, the underlying thesis of our work is that we must also be able to reason in terms of pattern instances — *groups* of objects and the interactions among them— to understand system behaviors from a design perspective. Since pi-traces record these interactions, they allow us to reason at the level of these groups.

A key property of design patterns is their flexibility, which enables the solutions they capture to be tailored to the needs of individual systems. At the same time, this flexibility creates opportunities for errors. Misunderstandings in how a pattern should be tailored for use can have disastrous consequences, especially when members of a design team have mutually incompatible views of the specialization details. The opportunity for misinterpretation can be greater in systems that involve complex object interconnections. Our approach enables us to specify the pattern requirements precisely while retaining its

flexibility; indeed, it often helps to identify *additional* dimensions of flexibility absent (or at least not apparent) from the standard informal descriptions. This was also a consideration in our earlier work, but expressing the contracts in terms of pi-traces further enhances flexibility.

Pattern-centric reasoning based on our formalism will help designers ensure the *design correctness* of their systems. Beyond the initial design phase, this type of reasoning will help ensure that *design integrity* is preserved *across* the lifecycle as the system design evolves. We illustrate our approach by developing the contract for the *Observer* pattern and apply it to reason about the behavior of a simple system built using the pattern.

**Paper Organization.** The rest of the paper is organized as follows. In the next section, we describe our approach to pattern contracts and subcontracts and the structure of pi-traces and how they are used in pattern contracts. In Section 3, we consider the subtleties in the *Observer* pattern and consider how we can develop a suitable contract to capture these subtleties. We also briefly consider the *Hospital* system built using this pattern and how we may reason about it. In Section 4, we discuss related work. In Section 5, we reiterate the surprising subtleties that arise in the use of even simple patterns in relatively simple systems, summarize our approach to developing pattern contracts using pi-traces, and consider possible directions for future work.

## 2 Contracts, Subcontracts, Pi-Traces

In our previous work [16, 9], we have proposed a *pattern contract language*. Since our focus in this paper is to investigate the problems in the use of patterns that arise because of complex object interconnections and how these may be addressed by the use of pi-traces, we will provide a brief overview of the structure of pattern contracts and subcontracts, referring the interested reader to our previous work for more complete details.

Most patterns consist of multiple *roles*. Each role consists of a *state* and a number of operations. Thus the *Observer* pattern contains two *roles*, Subject and Observer. In any *instance* of the pattern, a single object plays the role of Subject, and zero or more objects play the role of Observer; the observers are said to be *attached* to the subject<sup>1</sup>. The intent of the pattern is to ensure that whenever the state of the subject is *modified*, the observers attached to the subject are appropriately *updated* so that their states become *consistent* with the

new state of the subject. The pattern requires that the class playing the Subject role define a `Notify()` method, invoked when the state of the subject is modified. The `Notify()` method is in turn required to invoke the `Update()` method on each attached observer, which is in turn responsible for setting the observer’s state to a state consistent with the current state of the subject.

A pattern contract includes a *role contract* corresponding to each role of the pattern. The role contract for a particular role specifies the state of that role and specifies the behavior of each method of the role using standard pre- and post-conditions. We will refer to these methods as the “named” methods. Thus `Notify()` is one of the named methods of the Subject role of the *Observer* pattern. The role contract also includes an “other” specification. The need for this may be seen as follows. Consider a system  $S$  designed using the *Observer* pattern. The class  $C$  playing the Subject role in  $S$  will have to include methods that play the roles of each of the named methods of Subject, and these methods of  $C$  will have to satisfy the corresponding specifications included in the Subject role contract.  $C$  will, typically, also include *additional* methods depending on the needs of  $S$ . These other methods will be required to satisfy the other specification since otherwise the intent of the pattern may be violated even if the methods playing the roles of the named methods behaved according to their role contract specifications. The role contract also specifies the *enrollment* and *disenrollment* actions corresponding to this role; these identify the specific actions (such as calls to specific (named) methods or the construction of a new object) that represent the enrollment/disenrollment of an object to play the particular role in a particular instance of the pattern, and the conditions that must be satisfied at the time of the action.

Three additional items in the pattern contract are the auxiliary concepts block, the instantiation clause, and the pattern invariant clause. An *auxiliary concept* is a relation over two or more role states. An auxiliary concept is *used* in the pattern contract but it is not *defined* in the contract. Instead, definitions of each auxiliary concept, tailored to the needs of a particular system designed using the given pattern, are provided as part of the pattern subcontract for that system. Thus in the contract for the *Observer* pattern, one of the auxiliary concepts we use is *Consistent()* which is a relation over a state  $s_1$  of the subject and a state  $o_1$  of an observer and represents that the information in  $o_1$  is consistent with the state  $s_1$  of the subject; since the notion of consistency will depend on the system, it is appropriate to represent it as an auxiliary concept in the pattern contract.

<sup>1</sup>We use names starting with uppercase letters, such as Subject, for roles. We use corresponding lowercase names, such as subject, to refer to the objects that play these roles.

If  $P$  is a pattern and  $S$  a system designed using  $P$ , the  $P$ -subcontract for  $S$  defines the mappings that specifies which classes of  $S$  play the various roles of  $P$ ; for each class  $C$  that plays role  $R$ , which methods of  $C$  play the roles of the various named methods of  $R$ ; and the mapping of the state of  $C$  to the state of  $R$ . In effect, the subcontract tells us how the objects of  $S$  that are instances of the various classes of  $S$  may be *viewed as* objects playing the various roles of  $P$ . The subcontract also provides appropriate definitions for each of the auxiliary concepts used in the pattern contract. A *verification* task is to ensure that each method of a class  $C$  that maps to a specific named method of the role  $R$  meets the corresponding specification in the role-contract for  $R$  (under the state mapping defined in the subcontract). Another verification task is to check that all the *other* methods of  $C$  meet the other-specification of  $R$ 's role contract.

The *pattern invariant* is an assertion over the states of the various objects enrolled in any given instance of the pattern during execution (of a system designed using the pattern). For *Observer*, this would just state that the state of each observer attached to the subject is *consistent* with the state of the subject. The invariant will be satisfied if the classes playing the individual roles meet the requirements specified in the respective role contracts. The invariant is the essential behavior ensured by correct usage of the pattern.

The pattern invariant will not be satisfied at *all* times during the execution of the system. For example, if the subject state has been *modified* and we are still in the middle of the *notification* cycle, only some of the observers' states would be *consistent* with the current subject state. This can be problematic if, as in the scenarios we considered in the next section, the subject state keeps changing during the notification cycle because then the invariant will (almost) never hold! The problem is that an invariant of this kind, expressed just in terms of the states of the various objects enrolled in the pattern instance, is not adequate for systems where the object relations make such scenarios possible. We need to make use of the *pi-trace* to capture suitable conditions. Similarly, the specifications of the various methods in the role contracts would also need to be written making appropriate use of pi-traces. Before turning to these tasks in the next section, we first consider the structure of the pi-trace.

Suppose  $S$  is a system designed using a pattern  $P$ . With each instance,  $PI$ , of  $P$  that exists at runtime, we associate a *pattern-instance trace*,  $\pi\tau$  that will record every method call that any of the objects enrolled in  $PI$  participates in. Calls corresponding to instantiating the pattern as well as enrollments/disenrollments of objects

in roles are also recorded in  $\pi\tau$ . Each method call is recorded as two separate elements, the first corresponding to the invocation, the second corresponding to the completion of the call; of course, if, at a given point, the method call has not yet completed, only the element corresponding to the invocation will appear in  $\pi\tau$  at that point. Recording the method invocation and completion as two distinct elements in  $\pi\tau$  makes it possible to have additional elements between the two; these would correspond to nested method calls made during the execution of the first method that was called.

Consider now the information recorded in an individual element of  $\pi\tau$ . Suppose, we are currently in the middle of executing the body of a method  $m()$  applied on an object  $o1$  that is enrolled in  $PI$  and we call  $o2.n()$  where  $o2$  is also enrolled in  $PI$ . An element recording this invocation will then be appended to  $\pi\tau$ ; this element will include the identity,  $o2$ , of the target object, the identity,  $n()$ , of the method invoked, the values of any additional arguments passed to  $n()$ , *and* the current state of  $o1$ ; this final item is essential since it will play a key role in specifying the kinds of behavior that may occur as a result of the object relations in the system. Once control transfers to  $n()$ ,  $\pi\tau$  will continue to grow in the same manner, recording further method invocations that may be performed in the body of  $n()$ ; etc. Suppose, finally, that this call,  $o2.n()$  completes execution and control returns (to the calling method,  $o1.m()$ ). This completion will be recorded by appending another element to  $\pi\tau$ ; this element will include the value of any result returned *and* the state of  $o2$  at this time.

What if, in the scenario described above,  $o2$  was not enrolled in  $PI$ ? In that case, the invocation element will be recorded as before but the return element will not contain information about the state of  $o2$  since it is not relevant from the point of view of  $PI$ . What about the method invocations from within  $o2.n()$  and invocations that may be made from within *those* invocations etc.? As long as these invocations are on objects that are not enrolled in  $PI$ , none of them will be recorded on  $\pi\tau$ . But if one of them is on an object that is enrolled in  $PI$  then that invocation *will* be recorded on  $\pi\tau$  and we will be back to the situation described in the previous paragraph.

### 3 Observer Pattern

In this section, we will see how the use of pi-traces allow us to provide a flexible formalization of the *Observer* pattern. To see the kinds of issues that need to be addressed, consider the behavior of the `Notify()` method. Since this method is required to invoke `Update()` on each

attached observer, it is natural to require that the trace of `Notify()` contain corresponding invocations. But consider a more complex situation. Suppose that in a particular system, `s` is a subject object, and `o1`, `o2`, and `o3` are observers attached to `s`. During the execution of a method on `s`, the state of `s` is modified in a way that requires its observers to be updated; `Notify()` will be invoked. After completing the `Update()` on `o1`, `Notify()` will invoke `Update()` on `o2`. Now suppose that this invocation of `Update()` invokes a method `m()` on an unrelated object `o` (which `o2` references). Finally, suppose that `o` references the subject `s`, and that during the execution of `m()`, another method is invoked on `s`, resulting in the need for another notification round. To summarize, in the middle of a *notification* cycle, the state of the subject has been modified.

What does the correct usage of the pattern require at this point? Standard informal descriptions [8, 3] do not address this question directly, but it seems that another notification cycle is required to bring `o1` into a state consistent with the modified subject `s`. Suppose this nested notification cycle begins, and completes normally. Is the original notification cycle required to complete? If so, `o3` would be updated a second time, although it would already be consistent with the new state of `s`. More troubling is the state of `o2`. In the middle of its `Update()` call, due to an invocation on an object *external* to the pattern instance, the state of its subject `s` was further modified. What should be the state of `o2` when the original `Update()` completes? Should it be consistent with the state `s` was in when the invocation began, or the new state `s` has since taken on? One possibility would be to *forbid* such scenarios; and indeed, that is what our earlier formalism as well as other formalisms do [12, 7, 16]. But this type of scenario often arises naturally and hence we need to allow for it. We first outline a simple simulation of a hospital built using the *Observer* pattern in which the scenario arises, then consider how the pattern contract may be written to allow this.

There are three classes in the *Hospital* system, *Patient*, *Nurse* and *Doctor*. Instances of *Patient* play the Subject role; instances of the other two classes play the Observer role. Partial code for the *Patient* class appears in Fig. 1. This class will play the Subject role so it should include methods for attaching and detaching observers (doctors and nurses); we have omitted these. We have also omitted the field accessor methods that return information about the current state of the patient object. The `_nurses` and `_doctor` variable contain references to these observers. The `notify()` method calls `update()` on these objects. The other methods of

the class change the state of the patient and hence call `notify()` on these observers.

```

1 public class Patient {
2     private String _name;
3     private Integer _temp, _hrtRt, _medLvl;
4     private Set<Nurse> _nurses;
5     private Doctor _doctor;
6     ...addNurse(n), removeNurse(n)...
7     ...addDoctor(d), removeDoctor(...
8     public void adjustMeds(int medLvl)
9         { _medLvl = medLvl; notify(); }
10    public void adminCPR()
11        { _hrtRt=...; notify(); }
12    private void notify()
13        { ...call update() on nurses and doctor... }

```

**Fig. 1. Patient Class Code (partial)**

```

1 public class Nurse {
2     private HashMap<Patient,Integer>
3         _patientVitals;
4     public void update(Patient p) {
5         _patientVitals.put(p, p.getTemp());
6         /cm...call adjustMedLvl?... }
7     public String getStatus(Patient p) {
8         int t = _patientVitals.get(p);
9         if((t>90)&&(t<105)) return("good");
10        else return("bad"); } }
11 public class Doctor {
12     private HashMap<Patient,Integer>
13         _patientVitals;
14     ...constructors...
15     public void update(Patient p) {
16         _patientVitals.put(p, p.getHrtRt()); }
17     public void treat(Patient p) {
18         ...try some treatment; CPR?... } }

```

**Fig. 2. Nurse, Doctor Classes Code (partial)**

The *Nurse* and *Doctor* class maintain some information about the patient objects to which they are attached as observers<sup>2</sup>. The `treat()` method of the *Doctor* class (an *Observer*.other method) may cause a change in the state of the patient. Similarly, the `update()` method of the *Nurse* class may cause the state of the patient to change. These are the scenarios we described earlier in the paper.

Now consider the *Observer.Notify()* specification. Since the `update()` method of the *Nurse* class may *modify* the state of the patient, this may result in a notification cycle being initiated in the middle of a notification cycle. How do we write the specification of the method

<sup>2</sup>A given object may play roles in multiple pattern instances; here, if a given *nurse* may be attached to two or more *patients* and will act as an *observer* in each of the corresponding pattern instances.

to allow for situations such as this? The key is to write the specification as a set of requirements on the interactions between the various objects that are enrolled in the current pattern instance using the pi-trace,  $\pi\tau$ , to express the requirements since  $\pi\tau$  records all these interactions. Consider the following:

1. Suppose  $e1$  is the completion element of  $\pi\tau$  that indicates that control is back in `Notify()`. Suppose  $e2$  is the next element of  $\pi\tau$  (which will be the invocation element corresponding to the next method call made by `Notify()`). Then if the subject state recorded in  $e1$  is  $s1$  and that in  $e2$  is  $s2$ , the relation  $\neg\text{Modified}(s1, s2)$  must be satisfied. Note, however, that the subject state  $s1$  recorded in the completion element  $e1$  may be *modified* from that recorded in the corresponding (preceding) invocation element. That is because once control leaves `Notify()`, it is possible that the state of subject might change. But `Notify()` itself will not directly cause such a change; that is what the requirement involving the states recorded in  $e1, e2$  tells us.
2. The value of ``obs` in the states in  $e1$  and  $e2$  must be the same. Again, this means that `Notify()` cannot directly attach new observers or detach existing ones. But once control leaves `Notify()` changes are possible. In the *Hospital* system, a version of `Doctor.update()` may attach a new nurse to the patient depending on the current state of the patient. Such actions would be allowed by this requirement.
3. There exists  $k$  such that the subject state recorded in  $ek$ , the  $k^{\text{th}}$  element of  $\pi\tau$  and the state recorded in all of the subsequent elements of  $\pi\tau$  that involve the subject have the same value of ``obs`; each satisfies the  $\neg\text{Modified}()$  relation with the state in the next element of  $\pi\tau$ ; and beyond  $k^{\text{th}}$  element of  $\pi\tau$ , there is a call to `Update()` on each observer object to which there is a reference in the ``obs` value recorded in these elements. This clause ensures that all the observers that are attached to the subject when `Notify()` finishes will be updated to the current subject state. It allows any nested notification cycles to either complete thereby making unneeded `Update()` calls or shortcircuit those calls.

These requirements need to be expressed formally. To do so, we need to introduce appropriate functions and operations on  $\pi\tau$  that will allow us to extract, for example, the states recorded in particular elements; or classify some elements as invocation elements on certain objects and others as completion elements; etc. These notations as well as the specification of `Notify()` using them will be presented in a future paper.

In summary, the use of  $\pi\tau$  to express the requirements that `Notify()` must satisfy enables us to write a flexible specification for this method that, while capturing the essence of the behavior needed by the pattern, will be satisfied by many alternate ways that may be adopted by different systems for the purpose of notifying observers of a *modification* in the state of the subject.

## 4 Related Work

Various aspects of pattern formalization have been studied by other authors. Eden *et al.* [5] focus on capturing *static* properties of pattern implementations. Their approach relies on a higher-order logic formalism in which patterns are represented as formulae. The basic entities of the logic include functions and classes; formulae specify relations among them. The approach is well-suited to capturing properties that can be verified statically, but does not deal with the dynamic, *behavioral* properties of patterns.

Mikkonen's work [12] is closer to ours. His specification approach relies on an action system language similar to UNITY [4]. Pattern participants are represented as state tuples; guarded actions, selected for execution by a non-deterministic scheduler, are used to model their interactions. The approach supports layered composition and refinement; *safety* properties are protected through the application of *superposition* rules. But the specification actions do not directly correspond to method bodies, and the scheduling model hides control flow. Hence method trace requirements are not conveniently captured. The approach additionally lacks support for auxiliary concepts and pattern invariants.

Helm *et al.*'s [10] present a pattern contract formalism that has similarities to ours. Their approach captures static *and* dynamic properties. Each contract specifies the roles, member variables and method behaviors, object enrollment conditions, and a pattern invariant. The approach supports contract composition and refinement, and *conformance declarations* similar to our subcontracts. But there is no support for specifying constraints on auxiliary concepts which are required to prevent definitions that could violate pattern correctness. And there is no notion of an others specification. Finally, their contracts provide only limited support for specifying call sequence requirements. It is not, for example, possible to impose conditions on the *last* call to a particular method. Moreover, there is nothing analogous to our pi traces. Trace variables, of course, have a long history in the specification of distributed systems (e.g. [13, 11]), and more recently, the behavior of polymorphic method calls (e.g. [2, 15]). But the concept and application of a *pi trace* appears to be novel.

## 5 Conclusion

A popular quote from Beck and Cunningham [1] reminds us that "...no object is an island... objects stand in relationship to others, on whom they rely for services and control". This fact has a direct effect on reasoning about large OO systems. The key to such reasoning is to ensure that the trace of interactions among the objects has a central role in the specification and reasoning system. This applies especially to specifying design patterns because patterns are used extensively in building such systems. We introduced the notion of *pi-traces* for this purpose and showed, via the example of the classic *Observer* pattern, how the use of pi-traces allows us to provide precise and yet very flexible specifications for patterns. The use of pi-traces allows us to provide answers to such questions as, exactly what assertions are required to hold at each point during a system's execution if we are to be sure that the underlying design patterns have been implemented correctly? Assuming these assertions are satisfied, what system properties can we conclude as a result? By contrast, in previous work, the answers to these questions were either unclear or incomplete. The notion of pi-traces is, we believe, a fundamental contribution to the specification and reasoning literature.

Our future work will focus on *pattern validation* techniques. We plan to pursue two paths of exploration. The first will focus on the development of a formal verification system for proving the correctness of pattern implementations. The second will focus on the application of *automated* techniques for validating pi-trace requirements. Specifically, we plan to explore the application of (i) runtime monitoring tools, (ii) model checking techniques, and (iii) theorem proving tools.

## References

- [1] K. Beck and W. Cunningham. A laboratory for teaching oo thinking. In *The Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, October 1989.
- [2] M. Buchi and W. Weck. The greybox approach. Technical Report TUCS TR No. 297, Turku Centre for Computer Science, 1999. available at <http://www.tucs.abo.fi/>.
- [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 and J. Misra. *Parallel program design*. Addison-Wesley, 1988.
- [5] A. Eden. Formal specification of object-oriented design. In *Proc. Int. Conf. on Multidisciplinary Design in Engineering*, 2001.
- [6] A. Eden, A. Yehudai, and J. Gil. Precise specification and application of design patterns. In *Automated Software Engineering*, pages 143–152, 1997.
- [7] R France, D Kim, S Ghosh, and E Song. A uml-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30:193–206, 2004.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.
- [9] J.O. Hallstrom, N. Soundarajan, and B. Tyler. Amplifying the benefits of design patterns. In *The 9<sup>th</sup> International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 214–229. Springer, 2006.
- [10] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *OOPSLA-ECOOP*, pages 169–180, 1990.
- [11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [12] T. Mikkonen. Formalizing design patterns. In *Proceedings of 20th ICSE*, pages 115–124. IEEE Computer Society Press, 1998.
- [13] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE TSE*, 7:417–426, 1981.
- [14] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture: Patterns for concurrent and networked objects*. Wiley, 1996.
- [15] N. Soundarajan and S. Fridella. Understanding oo frameworks and applications. *Informatica*, 25:297–308, 2001.
- [16] 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 Softw. Eng. (ICSE)*, pages 666–675. IEEE Computer Society, 2004.
- [17] B Tyler, J Hallstrom, and N Soundarajan. A comparative study of monitoring tools for pattern-centric behavior. In M Hinchey, editor, *Proc. of 30th IEEE/NASA Software Engineering Workshop (SEW-30)*. IEEE-Computer Society, 2006.