Formalizing Design Patterns: A Comprehensive Contract for Composite

Jason O. Hallstrom
School of Computing
Clemson University
Clemson, SC 29634-0974
jasonoh@cs.clemson.edu

Neelam Soundarajan
Computer Science and Engineering
Ohio State University
Columbus, OH 43210-1277
neelam@cse.ohio-state.edu

ABSTRACT

Software patterns are used almost universally across design communities as the preferred mechanism for communicating best practice. And while the design archetypes captured by patterns continue to exert significant influence on software design decisions, there is no rigorous foundation for ensuring implementation correctness or reasoning about the systems in which patterns are applied. In this paper, we attempt to identify the conceptual elements necessary of any pattern formalism that satisfies these validation and reasoning objectives. We then present an overview of a particular pattern formalism developed as part of our prior work. The *Composite* pattern is used as a demonstrative example.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages, Methodologies*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Programming by contract, Reliability, Validation*

General Terms

Design, Documentation, Languages, Reliability, Verification

Keywords

Design patterns, pattern contracts, Composite pattern

1. INTRODUCTION

Design patterns began to gain adoption as a mechanism for disseminating best practice after the publication of the seminal *"Gang of Four"* (*GoF*) text [6]. Myriad pattern documentation efforts followed, resulting in a wide range of *pattern catalogs*. Representative efforts include the *"POSA"* series [2,3,9,12], and more specialized efforts devoted to particular implementation technologies (e.g., J2EE, .NET) [1,10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

While each of the patterns contained in these catalogs may not be universally accepted as best practice, one point seems beyond debate: After over a decade of use, patterns continue to exert a significant influence on the design of software, from standard desktop applications to embedded realtime systems and sensor networks.

There is little structural variation among pattern catalogs. Each adopts a variation of the stylized narrative format popularized by the GoF [6]. In this format, a pattern description consists of (*i*) a name, (*ii*) a problem (or objective), (*iii*) structural requirements expressed using UML (or UML-like) notations, (*iv*) code examples, and (*v*) supporting discussion elements. The last component may include a discussion of problem context, implementation pitfalls, system properties arising from a pattern's application, or other issues. And while there is no doubt that this documentation format has proven useful to practitioners, it is also inherently imprecise; patterns lack the foundation necessary to support rigorous validation and reasoning activities. Given the tremendous influence of patterns on software practice and the expectation that this influence will continue in the years ahead, software designers, implementers, and validators must have *precise* pattern specifications — specifications that enable them to reason rigorously about patterns and the systems in which they are applied.

In this paper, we present three contributions. (**C1**) First, we discuss the requisite features of a comprehensive specification formalism for software patterns. We do so by identifying the types of requirements that patterns impose and the dimensions of flexibility that must be preserved in documenting them. Flexibility is, after all, patterns' hallmark — a key contributor to their success and a focal point of our discussion. (**C2**) Second, we present an overview of a *pattern contract* formalism developed as part of our prior work [7,13]. The formalism supports specifications that are both precise and flexible and provides facilities for *pattern specialization*. These specialization facilities enable designers to capture commonly used pattern variants and to arrive at application-specific properties based on the patterns used in a given design. (**C3**) Finally, we apply the formalism to the *Composite* pattern [6][1,2]. Key benefits and limitations are discussed.

Paper Organization. Section 2 describes the require-

[1]Due to space constraints, we assume prior knowledge of the Composite pattern throughout the manuscript.

[2]The discussion is limited to sequential systems in the absence of object aliasing.

ments of a comprehensive pattern specification formalism. Section 3 summarizes our prior work on design pattern contracts. Section 4 presents the Composite pattern contract and discusses an associated specialization. Section 5 highlights elements of closely related work. Section 6 concludes with a discussion of limitations.

2. REQUIREMENTS ON PATTERN FORMALIZATION

On the one hand, it is clear that design patterns impose specific requirements on the classes that play their constituent *roles*. In the case of Composite, for example, it is clear that component, leaf, and composite objects are required to share a set of common interface elements and that each composite is responsible for dispatching calls to its children. On the other hand, it is also understood that patterns are intended to serve as reusable templates; they can be specialized as appropriate for particular scenarios. It is, for example, understood that the operations shared among participants in an instance of Composite will vary, as will the set of calls dispatched to the children of a composite object. This gets to the heart of the problem: An effective pattern formalism must balance the tension between descriptive precision and pattern flexibility. Here we identify the types of requirements imposed by patterns and the dimensions of flexibility that must be preserved.

Structural Requirements. Patterns impose structural requirements on participating objects. These include the roles that may participate in a pattern instance, the signatures that must be provided by objects playing these roles, and the inheritance and association relations among them. The classes that play the roles required by a given pattern will of course vary from one application to another, as will the method signatures they provide to satisfy their role responsibilities. The Leaf role, for instance, will be played by different classes in different applications of the pattern, and the signature of operation() will be implemented in an application-specific manner. Further, each class may provide *multiple* methods intended to play the part of operation(). Or more generally, multiple class methods may correspond to a single role method.

State Requirements. Patterns impose abstract state requirements on participating objects. Objects playing the Composite role, for example, must maintain a set of component objects (as children). It is understood, however, that this set may be implemented using any suitable realization.

Behavioral Requirements – State. Patterns impose behavioral requirements, expressed in terms of standard state-based pre-conditions and post-conditions. The addChild(c) method of Composite, for instance, requires that the component passed as argument not be a member of the composite's child set and ensures that it is added to this set upon termination. As is standard, these requirements can be satisfied in any manner the designer chooses.

Behavioral Requirements – Call Sequence. Patterns not only impose requirements on the state conditions that must be satisfied by particular methods, but also on *how* these conditions must be satisfied. These requirements are expressed in terms of *call sequence conditions* that must be respected during a method's execution. When operation() is invoked on a composite object, for example, it is generally required to place a similar call to its children. Another approach would be for the composite to traverse the tree structure (using getChild()) and invoke appropriate methods on each object that affect the same state changes. While the result would be identical, the implementation would violate a key pattern requirement.

Non-Interference Requirements. Finally, patterns impose implicit requirements on all *non-role* methods provided by participating classes. After all, a pattern describes a slice through a system; participating classes will generally provide method behaviors (and state elements) beyond those required to satisfy their role responsibilities. It is assumed that these behaviors will not interfere with pattern behaviors. A class playing the role of Composite, for example, might include additional (non-role) methods for interacting with the composite's children. These methods must not modify the child set or the intended behavior of the pattern will be compromised.

3. AN OVERVIEW OF A PATTERN CONTRACT FORMALISM

We now consider a pattern formalism designed to provide descriptive precision and pattern flexibility along the identified dimensions. We provide only a brief overview, referring the reader to [7, 13] for a more complete treatment.

In our approach, a pattern is represented by a *contract* that captures the requirements associated with using the pattern correctly and the behavioral guarantees that accrue as a result. *Specializations* of the pattern are represented by a *subcontract*. A subcontract refines a pattern contract to document the manner in which the associated pattern is tailored for use in a given system or to document a *sub-pattern* corresponding to a common usage of the pattern. In this way, contracts capture properties common to all applications of a pattern, while subcontracts capture properties specific to particular applications and sub-patterns.

3.1 Pattern Contracts

A contract consists of four main elements: *role contracts*, a *pattern invariant*, *state abstraction concepts*, and *interaction abstraction concepts*[3]. We describe each of these elements in the remainder of the subsection.

Role Contracts. The contract for a given pattern defines a role contract corresponding to each of the pattern's constituent roles. These specification entities form the core of a pattern specification. Each role contract specifies the abstract state elements, method behaviors, and non-interference conditions that must be satisfied by objects playing the associated roles[4]. The structure mirrors a standard interface specification: Each role contract specifies state elements, method signatures, and corresponding pre- and post-conditions. An additional set of post-conditions may be included to capture non-intereference conditions. These other conditions must be satisfied by all class methods that do not map to one of the role methods.

To specify call sequence requirements, we associate a *ghost variable*, τ (for *"trace"*), with each method invocation. Consider the invocation of a method $m()$. The instance of τ associated with this invocation records information about

[3]A pattern contract may additionally define pattern *instantiation* and *destruction* conditions. We omit these details.

[4]A role contract may additionally define role *enrollment* and *disenrollment* conditions. We omit these details.

the calls placed by $m()$ during its execution. More precisely, τ is an ordered sequence, with each entry corresponding to a single call. The entry records (*i*) the target object, (*ii*) the method invoked, and (*iii*) any argument values passed[5]. Call sequence requirements are then captured as conditions on τ, included as part of $m()$'s post-condition.

Pattern Invariant. A benefit of using many patterns is the behavioral guarantees they afford. Surprisingly, this is true even of *non-behavioral* patterns such as *Composite*, classified as a *structural pattern* by the GoF. As an example, in a standard application of the pattern, certain state conditions can be expected to hold across the nodes within a subtree based on the fact that calls are forwarded to child components. These guarantees are captured by the pattern contract in the form of a *pattern invariant* — a relation on the states of participating objects that holds at well-defined points in the system's execution[6].

Abstraction Concepts. While an application of Composite can be used to ensure that a particular state relation holds within a subtree, the contract cannot define the relation since the definition will vary from one application to another. Similarly, it would be overly-restrictive for the contract to specify the children to which an invocation of operation() must be forwarded since this, too, will vary. To provide this type of flexibility without sacrificing precision, pattern contracts declare *state abstraction concepts* and *interaction abstraction concepts*. The former is a relation on the states of participating objects, used in specifying the pattern invariant and the pre- and post-conditions included within the constituent role contracts. The latter is a relation on instances of τ, used in the specification of call sequence conditions. The key to the flexibility that these concepts provide is that while a pattern contract *declares* the concepts and imposes constraints on the allowable definitions, it defers the *definitions* of the concepts to the subcontracts associated with particular systems and sub-patterns.

3.2 Pattern Subcontracts

The purpose of a subcontract is to specialize a pattern contract so that the resulting specification captures a more specific version of the associated pattern. As such, a subcontract consists of specification elements used to document structural and behavioral refinements to a parent contract.

The first of these specification elements is a *role map*, used in two ways: The most common use is to document the mapping between a role specified in a pattern contract and an application class that plays the role in an application of the pattern. Or stated another way, a role map is used to specify the manner in which a class can be viewed as an instance of its role type. Alternatively, a role map may be used to document a mapping between two roles. In this case, the mapping captures the relationship between a general role and a more specialized version of that role used in a sub-pattern. In each case, a role map consists of a set of *state maps* and *method maps*. The former elements are used to document the realization of the state elements required by a

[5]In general, a more sophisticated trace mechanism is required to handle complex call sequence scenarios. We omit consideration of such scenarios.

[6]This relation always holds when control is outside of the objects participating in a pattern instance. While a much stronger guarantee is possible, space limitations preclude its consideration.

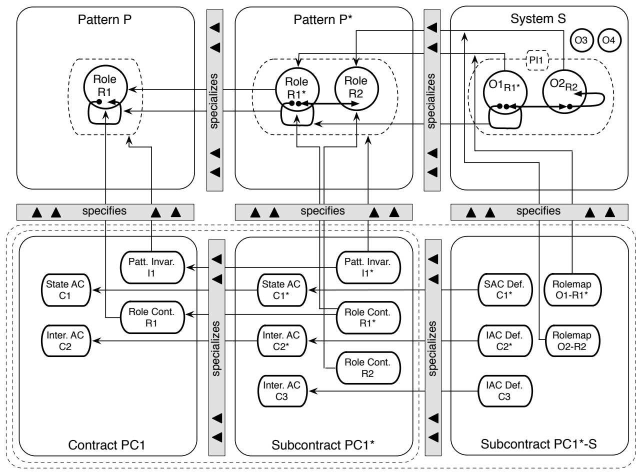

Figure 1: Contracts, Subcontracts, Specializations

role; they are analogous to abstraction functions. The latter elements are similar, but used to document method realizations. This includes documenting the mappings between signature elements, and argument and return values.

Finally, a subcontract specifies refinements to the abstraction concepts specified by the parent contract. These refinements may consist of concept definitions corresponding to a given application, or constraints on the allowable definitions. The latter are used to limit the definitions that my supplied to satisfy the requirements associated with a sub-pattern.

The relationships between patterns, sub-patterns, contracts, subcontracts, and applications are illustrated in Figure 1. In the figure, contract PC1 specifies pattern P. The role contract for R1 specifies implementation requirements on the role, and the pattern invariant I1 specifies an invariant across participating objects. Both are expressed in terms of the state abstraction concept C1 and the interaction abstraction concept C2. The sub-pattern P* specializes pattern P, as documented by subcontract PC1*. Note that this subcontract refines C1, C2, I1, and R1, and additionally adds a new role and a new interaction abstraction concept. Finally, the instance of sub-pattern P* in system S, PI1, is specified by subcontract PC1*-S. The subcontract provides definitions for C1*, C2*, and C3. It additionally provides rolemaps corresponding to objects O1 and O2, which play the roles R1 and R2, respectively.

4. COMPOSITE CONTRACT

We now apply the formalism to a common variant of the Composite pattern. For the sake of presentation, the contract has been segmented into separate listings. We describe each segment in turn.

The contract begins by declaring the state abstraction concepts used throughout the remainder of the document (Listing 1). The first, Modified(), captures the notion of a *significant change* to a composite object with respect to one of its children. At the point this concept is used, it determines the set of children that must receive a forwarded operation() call from a composite. More precisely, given the pre-conditional and post-conditional states of the target composite and one of its children, Modified() determines whether a call to operation() must be forwarded to the child.

```
pattern contract Composite {

 state abstraction concepts:
  Modified(Composite$_\alpha$, Composite$_\beta$, Component$_\gamma$)
  Consistent(Component$_\delta$, Component$_\epsilon$)
  constraints:
   $(\uparrow \alpha = \uparrow \beta) \wedge \neg((\uparrow \delta =$Leaf$) \wedge (\uparrow \epsilon =$Leaf$))\wedge$
    $\forall c1, c1^* \vdash$ Composite$, c2 \vdash$ Component ::
      ((Consistent$(c1, c2) \wedge \neg$Modified$(c1, c1^*, c2)$)
        $\Longrightarrow$ Consistent$(c1^*, c2)$)

 interaction abstraction concepts:
  ...omitted...

 pattern invariant:
  $\forall c1, c2 \vdash$ Component :
   $(c1 \in$players$) \wedge (c2 \in$players$)\wedge$
   $(\Uparrow c1 =$Component$) \wedge (c2 \in c1$.children$)$ :
     $((c2$.parent$= c1)\wedge$Consistent$(c1, c2))$
```

Listing 1: Composite Contract (part 1)

The second concept, Consistent(), is used to capture the notion of *state consistency* between a composite and a child. It is used in the post-condition of operation() to require that the method leave the target object in a state that is *consistent* with its parent. As we will see, it will also be used in expressing the pattern invariant.

The constraints clause restricts the concept definitions that may be supplied in a subcontract to ensure that the pattern invariant is satisfied. Three restrictions are imposed. First, the constraints require that the first two arguments of Modified() be of the same type (since this operation is only applied on two states of the same object in the contract). The "↑" notation denotes the application class (or specialized role) mapped to the target's type. Second, at least one of the arguments to Consistent must not be a leaf (since this concept captures consistency between a parent and a child — a relationship that cannot hold between two leafs.) Finally, the last conjunct requires that if two states of a composite are considered to be sufficiently similar according to Modified(), and the first is *consistent* with a given child, so too, must the second. This is necessary since the definition of Modified() controls whether operation() calls are forwarded — calls which are in turn responsible for ensuring *consistency* between parents and children.

For the sake of presentation, we provide a simplified contract, omitting interaction abstraction concepts.

Finally, the contract specifies the pattern invariant. If all implementation requirements are satisfied, Composite ensures that every child component is *consistent* —according to an appropriate definition— with its parent component.

Next, the contract specifies the role contract for the Component role (Listing 2). The notational elements within brackets indicate that exactly one class must be mapped to this role in an application of the pattern, and this class must be abstract.

The body of the role contract begins by requiring that classes playing the role maintain a Component reference, referred to as parent in the specification. As the name suggests, this variable is intended to store a reference to the component's parent, if any, in the composite tree[7].

[7]In general, it is more flexible to treat parent as a ghost variable, providing developers the ability to omit its realization.

```
role contract Component [1,abstract] {

 Component parent;

 void operation();
   pre: true
  post: (parent= #parent)∧
        Consistent(parent, this)

 others:
  post: (parent= #parent)∧
        (Consistent(parent,#this)
           ⟹ (Consistent(parent, this))
}
```

Listing 2: Composite Contract (part 2)

Next, the role contract provides the specification of operation(), and an others clause used to capture the conditions that must be satisfied by all *non-role* methods supplied by classes playing the role. The specification of operation() requires that the method preserve the parent reference and leave the target object in a state that is *consistent* with its parent. The non-intereference conditions are identical, but the consistency requirement is only imposed if the target was in a consistent state prior to the call to operation().

```
role contract Composite [+] : Component {

 Set<Component> children;

 void add(Component c);
   pre: c ∉ children
  post: (children= (#children∪{c}))∧
        (c.parent=this)∧
         ∀oc ⊢ Component :
          (oc ∈ #children) :
            ¬Modified(this, #this, oc)∧
        (|τ.c.operation| = 1)

 void remove(Component c);
   pre: c ∈ children
  post: (children= (#children−{c}))∧
        ∀oc ⊢ Component :
         (oc ∈ #children) :
           ¬Modified(this, #this, oc)

 ...other child management methods omited...

 void operation();
   pre: ...inherited from Component...
  post: ...inherited from Component...∧
        (children= #children)∧
        ∀c ⊢ Component :
         (c ∈children) :
           (Modified(this, #this, c)
             ⟹ (|τ.c.operation| = 1))

 others:
   ...inherited from Component...∧
   (children=children)∧
   ∀c ⊢ Component :
    (c ∈ #children) :
      ¬Modified(this, #this, c)
}
```

Listing 3: Composite Contract (part 3)

```
role contract Leaf [*] : Component {

 void operation();
  ...inherited from Component...

 others:
  ...inherited from Component...
}
```

Listing 4: Composite Contract (part 4)

The bulk of the contract is devoted to specifying the Composite role (Listing 3). The first line of the role contract indicates that one or more classes must be mapped to this role in an application of the pattern, and each must inherit from the class mapped to the Component role.

As before, the contract begins with state requirements: Participating classes must maintain a Set of component objects. This variable, children, stores references to each of the composite's children.

Next, the contract specifies the method behaviors required of composite objects: First, participating classes must supply child management methods. The pre-condition of add(), for example, requires that the child passed as argument not be contained within children. The method is required to add the child to children and assign itself as the child's parent. The next conjunct requires that the composite not be significantly modified (according to Modified()) by the call.

More interesting is the last conjunct, which specifies a call sequence requirement: $|\tau.\mathsf{c.operation}|$ denotes the subsequence obtained by projecting τ on object c and method operation(). Hence, the clause requires that the composite invoke operation() on the new child. While this requirement is not discussed in the original pattern description, it is essential to ensuring the pattern invariant; without it, there is no guarantee that the child will be in a state consistent with its parent. Requirements on remove() are analogous, but omit call sequence requirements. Other management methods have been elided.

The pre-condition on operation() is inherited from Component; it is trivially *true*. The inherited post-condition is strengthened: first, it requires that the children variable not be altered. More interestingly, it requires that if operation() modify the state of the component in a manner that is significant with respect to some child, the object is responsible for invoking operation() on that child. This ensures that if the original call breaks the pattern invariant, the forwarding behavior will re-assert the invariant.

The non-interference conditions specified in the others clause strengthen the conditions specified by the Component role contract. In particular, non-role methods of a class mapped to Composite are required to preserve the children variable. Further, they are not allowed to modify the state of the composite in a *significant* way.

Finally, the contract specifies the role contract for Leaf (Listing 4). The declaration indicates that zero or more classes may map to this role and each must inherit from the class mapped to Component. The remainder of the role contract is inherited without change.

To arrive at the implementation requirements and behavioral guarantees associated with a particular application of Composite, a corresponding subcontract must be specified. It is the composition of the subcontract and the contract that guides system implementation activities and assists in reasoning about *pattern-centric* behaviors. As an example, consider a standard application of the pattern in the context of designing a GUI library. Classes within the library might represent windows, frames, panels, and other graphical elements, and the tree structure imposed by Composite would mirror visual containment relationships. The subcontract for this application would provide role maps for each of the participating classes; the details are straightforward. More interesting are the concept definitions.

For simplicity, we assume that only one method plays the role of operation() — namely, a resize() method used to adjust the size of a visual container and all of its children. In this scenario, the definition of Modified() would rely only on the first two arguments: The relation would evaluate to *true* if the object states passed as argument had different width and height values, and $false$ otherwise. Similarly, the definition of Consistent() would evaluate to *true* if the component states passed as argument had equal dimensions, and $false$ otherwise. By substituting these definitions into the role contracts, application-specific requirements emerge. And by satisfying these requirements, developers are assured of the specialized pattern invariant: When control is outside of the participating objects, the dimensions of the children in any subtree total the dimensions of the parent. In this way, the contract formalism captures precise implementation requirements while affording flexible specialization to document applications and sub-patterns[8].

5. RELATED WORK

The benefits and pitfalls of pattern formalization have been discussed by other authors. A number of specification formalisms have been proposed. Here we briefly survey four representative efforts.

Eden and Hirshfeld [5] focus on specifying the *structural* (i.e., static) properties of design patterns. The authors describe a higher-order logic formalism in which patterns are specified as formulae. The basic terms of the logic consist of classes and methods. The associated relations correspond to standard syntactic concepts, including class membership, method invocation, and inheritance. Each pattern is specified as a list of participants (i.e., classes and methods) and the relations among them. While the approach handles rich structural properties, it does not provide facilities for state abstraction, pattern specialization, or behavioral properties.

In contrast to Eden and Hirshfeld's structural emphasis, Mikkonen [11] focuses on *behavioral* (i.e., dynamic) properties. Using his approach, patterns are expressed in an action system notation with roots in the UNITY [4] formalism for parallel and distributed systems. Each pattern is specified as a set of state elements, relations on these elements, and guarded assignments. Refinement is supported through *superposition*; specification layers can be composed without violating safety properties as long as each layer writes only to the state components it defines. While this approach offers a number of interesting benefits, including the ability to concisely specify complex temporal properties, it is, in a sense, *too abstract.* Structural and control-flow requirements are intentionally abstracted away, compromising the

[8]Sub-patterns are documented in an analogous matter; we omit there consideration.

efficacy of the formalism as a prescriptive mechanism for software design.

Taibi and Ngo [14] describe a hybrid approach. Their formalism relies on predicate logic to capture structural properties of patterns, and an action system notation to capture behavioral properties. In effect, their work combines the key elements proposed by Eden and Hirshfeld, and Mikkonen.

Interestingly, one of the most comprehensive notations for specifying patterns predates the GoF's text. Helm et al. [8] describe a notation for specifying *"behavioral compositions"* in object-oriented software, including those captured by patterns. There are a number of similarities with our work: A pattern is represented as a contract that specifies the participating roles, their required state elements and method behaviors, a pattern invariant, and instantiation conditions. The notation also provides facilities for contract composition and refinement and documenting the mappings between application classes and contract participants. While the underlying concepts seem essential, the realization lacks both precision and flexibility. Fundamental precision limitations include an inability to document method pre-conditions and a weakly expressive notation for documenting call sequence conditions. It is not, for example, possible to restrict the invocations made between calls or to relate the arguments and return values of successive calls. Fundamental flexibility limitations include the absence of state abstraction and the use of name-based (i.e., syntactic) mappings between application classes and contract participants.

6. CONCLUSION

We conclude by noting that the contract formalism presented here is an abbreviated version of a more complete specification notation. The same is true of the pattern contract for Composite. In particular, the trace mechanism has been simplified to accommodate space requirements, with important consequences on both the precision and flexibility of the resulting contract.

Consider, for example, the specification of operation() in Listing 3. The post-condition requires that if the method alters the state of the composite in a *significant* way (i.e., according to the definition of Modified()), the method must in turn invoke operation() on each affected child to re-assert the pattern invariant. But what if, after forwarding the appropriate calls, the method again modifies the state of the composite? Or alternatively, what if a call from the method re-enters the Component hierarchy above the executing node and modifies component state? And what happens if control is *always* within a participating object? When does the pattern invariant hold?

Addressing these questions hinges on the use of a more sophisticated trace mechanism. We have recently developed the concept of a *pattern-instance trace*, a behavioral projection on a single pattern instance, which we believe provides an elegant solution. We are currently experimenting with the notation and hope to discuss early results if invited for presentation at the workshop.

Acknowledgments

This work is supported by the National Science Foundation through awards CNS-0745846 and DUE-0633506.

7. REFERENCES

[1] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Mountain View, CA, USA, 2003.

[2] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

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

[5] A. Eden and Y. Hirshfeld. Principles in formal specification of object-oriented design and architecture. In *The 2001 Conference of the Centre for Advanced Studies on Collaborative Research*, pages (*cd–rom*), Indianapolis, IN, USA, November 2001. IBM Press.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[7] J. Hallstrom, N. Soundarajan, and B. Tyler. Amplifying the benefits of design patterns: From specification through implementation. In *Foundational Approaches to Software Engineering*, pages 214–229, Berlin, Germany, March 2006. Springer-Verlag.

[8] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–180, New York, NY, USA, October 1990. ACM.

[9] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, New York, NY, USA, 2004.

[10] Microsoft Corporation. *Enterprise Solution Patterns Using Microsoft .NET*. Microsoft Press, Redmond, WA, USA, 2003.

[11] T. Mikkonen. Formalizing design patterns. In *The 20thth International Conference on Software Engineering*, pages 115–124, Los Alamitos, CA, USA, April 1998. IEEE Computer Society.

[12] D. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[13] N. Soundarajan and J. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *The 26th International Conference on Software Engineering*, pages 666–675, Los Alamitos, CA, USA, May 2004. IEEE Computer Society.

[14] T. Taibi and D. Ngo. Formal specification of design patterns – a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.