SPECIFICATION AND RUNTIME MONITORING OF
OBJECT-ORIENTED SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Benjamin James Tyler, B.S., M.S.

* * * * *

The Ohio State University

2006

Dissertation Committee:

Neelam Soundarajan, Adviser
Paolo A.G. Sivilotti
Atanas Rountev

Approved by

__________________________
Adviser
Graduate Program in
Computer Science and
Engineering
ABSTRACT

The object-oriented (OO) approach has fundamentally changed the way we look at software, and has had an enormous influence on the software engineering discipline. Through the use of powerful tools such as inheritance and polymorphism, developers can specialize and enhance software systems while reusing large portions of already existing code. However, the use of these features can result in complex interactions and dependencies among parts of the code, which may make it difficult when trying to understand and reason about the system’s behavior. We will show how such difficulties can be addressed by the use of specifications which provide essential information about how the system can interact with external components. Although such specifications allow us to formally reason about systems in a modular way, to fully realize their benefits, we also need suitable software tools that can be used by general practitioners. Here, we present such a tool which allows for monitoring of systems at runtime to identify any violations of these specifications. The tool exploits the same mechanisms of inheritance and polymorphism, and does not require the availability of the system source code. We will also look at how a similar approach can be applied to track the execution flow through OO class hierarchies.

Design patterns, a progression from class-level to design-level OO reuse, have also had a large impact on the way we think about and design large software systems. Not
only do patterns allow for the reuse of time-tested solutions, but they also enable others studying the system in question to get a deeper understanding of how the system is built and why it behaves in particular ways. Design patterns have traditionally been presented in an informal manner, however, and such descriptions often contain ambiguities that can lead to different interpretations and assumptions made about specific parts of the system. This can result in misunderstandings among developers working on the same project, and can eventually manifest themselves as incompatibilities among different parts of the system. Hence, we again need ways to precisely specify behaviors of patterns and systems built using them, as well as software tools to monitor the systems to identify any violations of the specifications. We present a pattern contract language that addresses the specificity of individual applications, while maintaining the flexibility inherent in the design patterns. We then present a tool which generates aspect-oriented monitors for checking pattern contract violations during runtime.
To my wife and family.
ACKNOWLEDGMENTS

First, I must thank Neelam Soundarajan, for all of his guidance, encouragement, and most of all, patience through my years of graduate study in Computer Science. I have learned so much from him, both from a professional and a personal standpoint. I am truly grateful to have him as my advisor.

I would also like to thank Steve Fridella and Jason Hallstrom, for laying much of the groundwork for my own research, and also for just being all-around great guys.

A special thank-you goes to my old advisor from the Mathematics Department, Randall Dougherty. He reminded me why math is so interesting, and without him, I probably would have taken a very different path in life.

I am also grateful for the folks involved with the Reusable Software Research Group: Bruce Weide, Tim Long, Wayne Heym, Bill Ogden, Paolo Bucci, Joan Krone, Nigamanth Sridhar, Scott and Hilary Pike, Santosh Kumar, Chris Bohn, and Mike Gibas. The RSRGers have provided a certain “family atmosphere” during my years as a graduate student in Computer Science—always helpful and supportive, while not shying away from giving me constructive criticism when I needed it. I want to single out Bruce Weide, Tim Long, and Joan Krone for all of their help and advice regarding my professional development.
I would also like to thank Paul Sivilotti, for teaching me how to write proofs that are both rigorous and understandable by human beings. I also thank him for his detailed comments on earlier drafts of this dissertation.

For their moral support and friendship through the years, I thank Jeremy and Karen Schmersal, Scott Simms, Matt Pohlman, Ted McCoy, Cosmin and Nicoleta Roman, and Rich and Mariana Sharp. My time here in Columbus has been so much more enjoyable because of all of you, and I will always treasure our shared memories.

I give my thanks and love to my family: my parents Leonard and Diane Tyler, and my sister and brother-in-law Shannon and Sean Robinson. Without your encouragement and belief in me, I would not have been able to have made it this far.

And finally, I must thank my wife, Almira, who seems to be the only person in the world at times to fully understand me—my hopes, fears, and dreams—and still willingly goes along with me on the journey. You are my Dearest, and my love.
VITA

August 3, 1973 .................................... Born – Tawas, Michigan

March 2000 ....................................... M.S., Computer and Information Science
The Ohio State University

December 1998 ................................. M.S., Mathematics
Thesis: Exogenous Logics of Programs
Adviser: Prof. Randall Dougherty
The Ohio State University

May 1995 ......................................... B.S., Mathematics
Michigan Technological University

September 2000 – August 2005 ............ Graduate Teaching Associate
Computer Science and Engineering
The Ohio State University

September 1995 – August 2000 ............ Graduate Teaching Associate
Mathematics
The Ohio State University

PUBLICATIONS

Research Publications

Benjamin Tyler, Jason O. Hallstrom, and Neelam Soundarajan. A comparative study
of monitoring tools for pattern-centric behavior. To appear in Proceedings of the 30th

Jason O. Hallstrom, Neelam Soundarajan, and Benjamin Tyler. Amplifying the
benefits of design patterns: From specification through implementation. To appear


FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

<table>
<thead>
<tr>
<th>Field</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Methodology</td>
<td>Prof. Neelam Soundarajan</td>
</tr>
<tr>
<td>Theory of Computation</td>
<td>Prof. Rephael Wenger</td>
</tr>
<tr>
<td></td>
<td>Prof. Kenneth Supowit</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
<td>Prof. DeLiang Wang</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td><strong>Chapters:</strong></td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Reuse and Object-Oriented Development</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Monitoring Object Interactions</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 Formalization and Monitoring of Design Patterns</td>
<td>6</td>
</tr>
<tr>
<td>1.3 The Approach</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Thesis Statement</td>
<td>12</td>
</tr>
<tr>
<td>1.5 Contributions and Assessment</td>
<td>12</td>
</tr>
<tr>
<td>1.6 Organization of the Dissertation</td>
<td>13</td>
</tr>
<tr>
<td>2. Specifying Object-Oriented Systems</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Problem Overview</td>
<td>15</td>
</tr>
<tr>
<td>2.2 A Simple Example</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Reasoning with Black-box Specifications</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Abstract and Concrete Specifications</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Specifying the <strong>Order</strong> Class</td>
<td>21</td>
</tr>
</tbody>
</table>
# Table of Contents

2.4 Trace-based Specifications ........................................ 24  
2.5 Related Approaches ............................................. 33

3. Monitoring Grey-box Specifications .............................. 35  
3.1 Problem Overview ................................................. 35  
3.2 Object-Oriented Frameworks ..................................... 36  
3.2.1 An Example: Diagram Editor ................................. 38  
3.2.2 Diagram Editor Implementation ............................ 44  
3.2.3 Specifying the Diagram Editor ............................. 51  
3.3 Monitoring Trace-Based Specifications ....................... 58  
3.3.1 Monitoring with Trace-Saving Classes ..................... 60  
3.3.2 Walking through a Monitoring Run ....................... 76  
3.3.3 Running TCGen and Sample Output ....................... 80  
3.4 Related Work .................................................... 85

4. Monitoring Control-Flow Through Inheritance Hierarchies ... 89  
4.1 The Yo-Yo Problem ............................................... 90  
4.2 Methodology Overview .......................................... 94  
4.3 Tracking the Method Calls ..................................... 96  
4.3.1 Down-calls .................................................. 98  
4.3.2 Up-calls .................................................... 101  
4.4 Running PolyTracker and Sample Output ..................... 104  
4.5 Discussion ...................................................... 109  
4.6 Related Work .................................................... 111

5. Specifying Design Pattern Contracts ............................ 113  
5.1 Introduction ..................................................... 114  
5.2 Flexibility and Specificity in Pattern Contracts .......... 116  
5.3 The Pattern Contract Language ................................ 118  
5.3.1 Pattern Contract Grammar ................................ 118  
5.3.2 Subcontracts ............................................... 126  
5.4 Pattern Contract and Subcontract Examples .................. 129  
5.4.1 Observer Pattern Contract ................................. 129  
5.4.2 Hospital Application and Subcontract .................... 137  
5.4.3 Memento ................................................... 143  
5.4.4 Chain of Responsibility .................................. 149  
5.5 Behavioral v. Non-Behavioral Patterns ....................... 152  
5.6 Related Work .................................................... 153
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Order class code</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>BPOrder class code</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Black-box specifications for Order</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Trace-based specifications for Order (preliminary)</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Grey-box view of t()’s interactions</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>An alternative trace-based postcondition for Order\textunderscore combine()</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>A diagram editor in use</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Diagram editor framework</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Node class code</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Edge class code</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>DiagEditor class code (part 1)</td>
<td>47</td>
</tr>
<tr>
<td>3.6</td>
<td>DiagEditor class code (part 2)</td>
<td>49</td>
</tr>
<tr>
<td>3.7</td>
<td>DiagEditor class code (part 3)</td>
<td>50</td>
</tr>
<tr>
<td>3.8</td>
<td>Specifications for the DiagEditor add methods</td>
<td>52</td>
</tr>
<tr>
<td>3.9</td>
<td>Specification for moveNode()</td>
<td>56</td>
</tr>
</tbody>
</table>
6.12 Creation of after advice ........................................ 174
6.13 Parse tree for ⟨subContract⟩ ................................. 176
6.14 Subtree for ⟨roleMap⟩, highlighting ⟨stateMap⟩ subtree .... 178
6.15 Parse subtree for ⟨auxConceptDefs⟩ ........................ 179
6.16 The Observer_Monitor contract monitor (partial) ........ 181
6.17 after advice generated for Update() role method ........... 183
6.18 Monitor size v. contract size (in non-comment tokens) ..... 184
6.19 Runtime overhead (in m/s) ..................................... 184
6.20 Performance comparison data ................................. 189
CHAPTER 1

INTRODUCTION

Object-oriented (OO) programming provides powerful tools in the form of inheritance, polymorphism, etc. But programs that exploit these can contain subtle bugs. Hence we need ways to precisely specify the behaviors of these programs and software tools that enable us to monitor the programs at runtime to see if any specifications are violated. Design patterns are another set of powerful and important tools in the OO designer’s tool box. Again systems built using patterns, especially large systems built by teams rather than by individual software developers, can contain subtle bugs. Hence, again we need ways to precisely specify behaviors of patterns and systems built using them; and software tools to monitor the systems at runtime to identify any violations of the specifications. This thesis presents work toward developing such specification techniques and monitoring tools based on these specification techniques.

1.1 Reuse and Object-Oriented Development

With our ever increasing use of and consequent reliance on computers in today’s world, software engineers are faced with the challenge of creating software systems that are reliable, while keeping within time and budget constraints. One common theme, the theme of reuse, is found in many strategies employed by software engineers
to tackle their difficult task [65, 74, 33, 5, 12]. Whether it is reuse of code, design principles, design practices, test cases, or tools, the effort to develop new artifacts is lessened, while repeated use (hopefully) will lead to their further refinement.

One major development has been the introduction of object-oriented programming [60]. The OO paradigm allows developers to view software as a system of interacting, yet autonomous entities. These entities, which are referred to as objects, are characterized not only by what data they contain, but also by what operations can be performed by the objects themselves. These “object characterizations”, or classes, can be (re)used in different programs or systems, reducing the programming effort needed. Widely-used OO languages such as C++ and Java have a large selection of already existing classes in their libraries. However, what sets OO apart from other class-based languages is the ability to create new classes of objects through specializing already existing classes, and the use of polymorphism\(^1\). These powerful mechanisms allow programmers to enhance classes without having to rewrite class code that already suits their needs.

A more recent development, design patterns [7, 33, 43, 69], has also had a large impact on the way we think about and design large software systems. Design patterns are descriptions of solutions to commonly occurring problems in object-oriented software design, and can be found in pattern catalogs such as [33, 2, 18]. Grady Booch states that the promise of reuse through the use of objects was only fully realized through design patterns—reuse at the level of design itself [12]. Not only do patterns allow for the reuse of time-tested solutions, but they also enable others studying the system in question to get a deeper understanding of how the system is built and

\(^1\)Throughout this thesis, by “polymorphism” we mean inclusion or subtype polymorphism [19] achieved in standard OO languages such as Java via inheritance and dynamic binding.
why it behaves in particular ways. Furthermore, as the system evolves over time, the patterns used in the development of the system provide guidance on managing the evolution so that the system remains faithful to its intended design.

1.2 Problem Statement

Unfortunately, innovations such as object-orientation and design patterns do not come without cost. Language features such as inheritance and polymorphism introduced by OO programming languages, while allowing for more reuse, allow for complex interactions and dependencies among parts of the code that may make it difficult when trying to reason about system behavior. Design patterns have attempted to help keep some of this complexity under control by outlining some “best practices” in software design, but they have been, for the most part, presented in an informal manner. Such informal descriptions often contain ambiguities that can lead to different interpretations and assumptions made about specific parts of the system. This, in turn can result in misunderstandings among developers, which can be very problematic for team-based projects, where these different interpretations can manifest themselves as incompatibilities among different parts of the system.

To fully realize the benefits of object-orientation and design patterns, two requirements must be met:

1. We must have ways to characterize intended system behaviors precisely, including ways to formally specify interactions between system components as well as the conditions that a system must satisfy with regard to the structure of the classes and the behaviors of particular methods of the classes. We also must be able to specify the behaviors that the resulting system is guaranteed to exhibit.
2. We need suitable tools that designers can use to check whether their systems do, in fact, satisfy such specifications mentioned in item 1.

Although formal specification enables the formulation of correctness proofs for software systems using mathematical reasoning, this is usually not done in practice. Formally verifying system correctness can be very tedious, and few practitioners have the necessary training to do such formal verification. However, formal specifications are still essential; without them, there is too much room for ambiguity and misunderstanding among developers and maintainers when informally reasoning about the system.

What is used extensively in practice in lieu of formal reasoning, is testing and tool-facilitated analysis. Software tools that automate large parts of the testing and analysis process do not require such specialized training to use, and do not require near the amount of human guidance needed by formal verification tools. Here, specifications used by such tools must be written formally in order to be parsed and processed by the tool. Before we can reap the benefits of automated testing and analysis, however, proper tools must exist that are able to process these specifications of critical behaviors, and evaluate whether or not system behaviors observed during runtime follow these specifications.

1.2.1 Monitoring Object Interactions

One of the most common tool-supported methods of gaining assurance that programs behave the way they should is testing. Although formal verification tools can give guarantees that software systems adhere to their specifications, it requires individuals with specialized training to guide such tools through much of the process of
performing correctness proofs. The testing process, however, can be largely automated with the proper tools, and they generally do not require such specialized skills to use.

*Black-box testing* [8] involves checking if methods of a class, when given certain inputs that satisfy their preconditions, give outputs that adhere to their postconditions after execution. Tools such as the widely used JUnit [6] help facilitate such testing, and are freely available to the public. However, the behaviors that characterize essential features of OO and many design patterns cannot be tested in a black-box manner using such tools. As previously mentioned, these behaviors include interactions between objects which would need to be monitored *during execution* in order to test for them; black-box testing relies only on checking the resulting program state *after execution* is completed. These difficulties are recognized as some of the biggest problems in testing OO software [9, 58].

What is needed is an approach that allows us to monitor the system during execution. This could be done through *code instrumentation*, where special code is inserted into the system code at specific points to allow for the monitoring apparatus to observe what is occurring at these points. Such an approach does modify the system, and if not done carefully, can introduce new bugs. Using tools that automate the instrumentation process is preferable to instrumenting code by hand, which can be time-consuming, tedious, and as a result, error-prone. Instrumentation, however, cannot be done in the first place if we do not have the source code for the system components that are to be monitored, which is often the case when they are purchased from a vendor. For this reason, it has been argued that *built-in testing* code be a required part of the components themselves [34]. This does shift an additional burden
on the people coding the components, while not improving their base functionality—such code generally adversely affects the component’s efficiency. In any case, many of today’s components, as well as most legacy systems, do not utilize such built-in testing code.

1.2.2 Formalization and Monitoring of Design Patterns

When formalizing design patterns, one potential risk is that flexibility, which is the hallmark of design patterns, may be compromised, and will no longer be applicable to specific systems [68]. Consider the Observer pattern [33]. From the standard informal descriptions of the pattern, a designer can see that it is intended to be used when one or more observer objects are interested in observing the state of a subject object. When an object wishes to become an observer of a subject, it will invoke the Attach() method of the subject; and when it is no longer interested in that subject, it invokes its Detach() method. If such notions are stated formally, could we then apply the pattern to systems that, for instance, do not specifically have a method called Attach() that resides in the subject’s class? What about systems that have observers that can be interested in more than one subject, or even systems that allow multiple subjects in the first place?

The description of the pattern in [33] further states, “... subject notifies its observers [by calling Notify()] whenever a change occurs that could make its observers’ state inconsistent with its own”. What is not clear from the description is how the subject will know when its state has become inconsistent with that of one or more observers? Indeed, what does it mean to say that the subject state is inconsistent with an observer state? The risk is that if, for example, we adopt one definition for
the notion of *inconsistency*, the pattern may not be usable in systems that have a different notion of this concept.

Clearly, an effective specification approach needs to be expressive enough to capture the key features of design patterns. From the above description alone, we see that such specifications need to be able to convey information such as “object A invokes method m() on object B under condition C”. A more problematic issue is that the specification approach should address the *specificity* of individual applications, while maintaining the *flexibility* inherent in the design patterns. Notions such as *inconsistency* which may be different from application to application, are critical to the understanding of the pattern and need to play a part in the specification. At the same time, how can we verify that a specific application implements a pattern correctly, when the specification uses such general notions?

If we wish to have tool support for runtime checking of pattern violations, these problems become even more obvious. What is needed is a concrete way of determining if a given property does or does not hold for a specific system—general notions alone are not adequate. But, not only do we have to contend with evaluating things such as *inconsistency* in a monitoring approach, we also have to address difficulties such as *Subject* classes that contain methods that behave like *Attach()* but are not named *Attach*. Because design patterns often involve behaviors that encompass several objects of different classes, a tool that checks pattern violations must be able to monitor these objects simultaneously and integrate the collected information. The monitoring process is further complicated by the presence of multiple instances of the same, or even different design patterns.
1.3 The Approach

Our approach to handling these problems is the formulation of specification constructs and languages which are expressive enough to capture vital OO system behaviors, and the development of automated software tools that can generate monitoring code for such specifications. One theme that is found throughout this work is the need to specify certain grey-box behaviors [4, 16, 32]. Grey-box behaviors encompass the interactions among objects within a system, in addition to the functional behaviors described in standard black-box specifications. Whether you are trying to capture complex call patterns made by a particular method, track the flow of control in a deep inheritance hierarchy, or express that a particular system design explicitly requires certain interactions, there is a necessity for such specifications. By noting that such interactions are generally precipitated by objects invoking methods on other objects, we characterize such interactions by using trace variables which specify sequences of calls made by a particular method. Previous work by Soundarajan and Fridella [77, 78] provide a basis for the use of traces, which is discussed in Chapter 2.

Given system specifications that include trace variables, we would like to be able to monitor them during runtime. Furthermore, it would be ideal if such monitoring could be done without code instrumentation or access to the source code at all. To accomplish these goals, we leverage the same OO mechanism that makes reasoning about OO systems difficult, polymorphism. Specifically, we create polymorphic wrappers for the particular objects whose method calls need to be tracked and recorded on the trace variable. The wrapper classes extend the classes to be monitored, and override the methods to be tracked using special code to store the necessary trace
information before and after the original method in the base class is invoked. Polymorphism allows us to use these wrapped objects in place of the original objects, and dynamic dispatch assures us that the wrapper code will be invoked at the proper times. None of this requires the source code of the classes being monitored being available. In Chapter 3, we present the tool TCGen, short for Test Class Generator, which automatically generates these wrappers from trace-based specifications, along with the support code for saving traces, and test drivers that can check trace-based assertions.

Another characterization of object interactions, which stresses the flow of control through OO inheritance hierarchies, is the yo-yo graph [82, 9]. Instead of stating what methods a particular method calls directly, as in traces, a yo-yo graph shows the chain of calls initiated by calling a single method. Yo-yo graphs are different from sequence diagrams of the Unified Modeling Language [14], in that they focus on the calls made upon a single object within its own classes, and they explicitly illustrate where in the inheritance hierarchy these calls are dispatched to, either through dynamic dispatch or through super-calls. Yo-yo graphs are so called, because they tend to bounce up and down in illustrating these calls. In Chapter 4, we present a tool, PolyTracker, that we have created that allows for the tracking of such calls, and the display of the resulting yo-yo graph. PolyTracker uses the same principles as TCGen in tracking method calls, and when coupled with the ability to check pre- and postconditions, it provides an effective way to isolate the location of bugs that may be buried in the code of parent classes.

Let us now turn to the higher-level behaviors described in design patterns. We have already mentioned how specifying them presents many additional challenges,
including the problem of how to precisely express critical behaviors while not losing the ability to apply them to a variety of systems that use them. To address these specification concerns, we have developed the Pattern Contract Language, or PCL for short. PCL can also be used to generate runtime contract monitors, which we will describe shortly. PCL solves the problem of generality versus specificity by separating specifications into two parts: pattern contracts, and application subcontracts.

Pattern contracts specify what is general: they name the roles played by objects in a specific pattern, state invariants that must hold during execution, and give descriptions of the methods that need to be present in terms of their pre- and postconditions. One pattern contract for a given design pattern is applicable to all systems that use the pattern. However, to retain generality, the assertions in the contract can contain expressions using auxiliary concepts, which are relations whose precise definitions are not given in the contract. For example, in the Observer pattern, the notion of consistency is expressed in the contract using an auxiliary concept relation Consistent on subjects and observers, while the definition itself for Consistent is not given.

Subcontracts\(^2\) are used to specify how a particular contract is applied to a specific system. They provide the mappings from system methods to methods in the contract, and from system state to the abstract state used in the contract. This is also where concrete definitions for the auxiliary concepts are given. Pattern contracts can be used in different contexts for different systems, whereas subcontracts can only be applied to specific systems.

\(^2\)We chose the term “subcontract” to emphasize that they extend and specialize (pattern) contracts, much in the same way subclasses and subaspects extend and specialize classes and aspects. The term here should not be taken as in the usual sense, where a larger project is subdivided into smaller ones.
One of the guiding principles of the work is to make specifications and their associated tools easy to use for general practitioners. It is true that the formal specifications in the PCL contracts may be difficult for some to write, but this is not a problem because they will only have to be written once, and can be reused for the different systems that incorporate them. One of the future goals of this work is to store these pattern contracts in an online catalog and make them available to the general public, so only a handful of people would have to write them. As for the subcontracts, which will have to be written by practitioners, they are considerably easier to write, and do not require the use of mathematical expressions or logic.

Given PCL specifications for a given system, we can also generate pattern contract monitors from contracts and subcontracts that check whether a system violates the pattern contract during runtime. In Chapter 6, we describe the tool MonGen, which is short for Monitor Generator, that automates this generation task. The monitors that MonGen creates are written in the aspect-oriented programming (AOP) [49] language, AspectJ [47]. AOP allows for cross-cutting concerns—interrelated behaviors or features that can or should be found in (possibly) different parts of a system—to be coded in a single module, called an aspect. Using a special weaving compiler, the code in the aspect or aspects will be inserted into the proper places in the object code produced for each of the system classes. For our purposes, the pattern contract monitoring code is contained within the aspects, separate from the system classes that implement the pattern. Because many of the behaviors associated with design patterns often involve several components of a system, AOP aspects provide an effective means by which runtime checking of pattern specifications can be done. By compiling
the monitoring aspects generated by MonGen in with the system code using the AspectJ weaving compiler, the resulting system will exhibit the same behavior as before with the pattern checking code “built in”. This allows runtime monitoring of the PCL contracts for a system simply by running test cases on the system. MonGen is also able to handle checking of multiple patterns and pattern instances within the same system. Details of MonGen and the aspects it generates will be covered in Chapter 6.

1.4 Thesis Statement

The thesis underlying the work presented in this dissertation can be summarized in the following two parts.

- An approach to grey-box specification that captures information about object interactions provides an effective way to reason about components of object-oriented systems in a modular way. Furthermore, a process of generating checking code for such specifications can be automated without relying on the source code of the components under test.

- An abstract specification approach that identifies object roles and their associated behaviors, including the interactions among the roles, can be used to precisely specify design patterns in object-oriented programs. Furthermore, an automated process of generating aspect-oriented monitoring code provides an effective means for checking such specifications.

1.5 Contributions and Assessment

The work found in the dissertation makes the following principal contributions:
• A monitoring technique that allows for trace information to be captured during runtime, where the source code of the components being monitored is not required. A tool, TCGen, is presented which automates the generation of polymorphic wrappers from trace-based specifications. These wrappers contain the necessary code for capturing traces, as well as checking code that evaluates trace-based specifications. Another tool will be presented, PolyTracker, which illustrates that similar principles can be used to track the flow of control through the code in OO inheritance hierarchies.

• A specification language for design pattern contracts, PCL, which allows for the precise specification of the critical behaviors found in patterns, while allowing enough flexibility so that the pattern can be applied to different systems. We will illustrate the effectiveness and flexibility of PCL by formalizing three design patterns from the literature.

• A monitoring tool, MonGen, which generates aspect-oriented monitors from PCL pattern contracts. We will provide an in-depth discussion of how MonGen and the monitors it creates work, as well as demonstrate how these monitors allow for the effective runtime checking of pattern contracts through the use of examples. We will also provide a preliminary analysis of the amount of coding effort saved in using MonGen, as well as the runtime overhead incurred by using the monitors.

1.6 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides background on specifying object-oriented systems, and about the need for grey-box specifications
for modular reasoning. Chapter 3 continues the discussion on grey-box specification, and illustrates why trace information is needed to capture the key features of OO frameworks. In this chapter, we present the TCGen tool, and show how it can be used to capture call traces, and illustrate its use with a case study. Chapter 4 presents how the same ideas used in TCGen can be used to capture the call chains associated with yo-yo graphs. Here, the PolyTracker tool will be introduced, and we will go through an example showing its use.

In Chapter 5, we introduce the pattern contract language PCL. We first present the syntax and semantics of the language, and then illustrate its flexibility using several patterns and pattern applications as examples. In Chapter 6, we present the MonGen tool, and discuss how it addresses some of the difficulties in monitoring pattern contracts. Later in the chapter, we show its use on the examples presented in Chapter 5, and then conclude with a brief analysis. We will examine related work for each of the topics covered in Chapters 2–6 at the end of each of these chapters. We conclude the dissertation in Chapter 7, which summarizes the contributions of the work, and presents some avenues for further study.
CHAPTER 2

SPECIFYING OBJECT-ORIENTED SYSTEMS

This chapter is primarily background on the difficulties with, and possible approaches to, specifying object-oriented systems. We begin by demonstrating how the features of OO languages can render standard black-box specifications inadequate for program understanding and modular reasoning. We then proceed by showing how these difficulties can be overcome by the use of traces, a technique for specifying interactions between objects. Before we present a simple illustrative example in Sec. 2.2, we first give a brief overview of the problem, as well as introduce some terminology.

2.1 Problem Overview

Suppose that C is an OO class, and D is a derived class of C. D may redefine some of the methods of C, while inheriting others directly from C without change. D may also introduce some new methods. Our focus will be on those methods inherited by D that may exhibit behavior that is different from their behavior in the base class C because of calls to methods that are redefined in D. The design patterns literature [33] refers to these inherited methods as template methods, and the methods that the template methods may invoke that may be redefined in D are called hook methods.
Let $t()$ be a template method of $C$, and $h()$ be a hook method that $t()$ invokes. As we just noted, redefining $h()$ in the derived class enriches also the behavior of $t()$. When reasoning about $t()$ in the base class, we would have appealed to the base class specification of $h()$ to account for the effects of the calls that $t()$ makes to $h()$. In order to be sure that the conclusions we have reached about the behavior of $t()$ apply also to its behavior in the derived class despite the redefinition of $h()$, we have to require that the redefined $h()$ satisfies its base class specification. This requirement is the essence of behavioral subtyping [56, 23]. However, ensuring that $t()$ continues to behave in a way that is consistent with its base class specification is only part of our concern. The reason that we redefined $h()$ in the derived class was to enrich not only its own behavior, but also the behavior of other methods such as $t()$ that call it, as we will see in the simple example in Sec. 2.2. The challenge to understanding and reasoning about the behavior of $t()$ comes from its interaction with enrichable methods such as $h()$. This task is made much more difficult when the source code for the base class $C$, and hence the template method $t()$, is unavailable. This is often the case when developers purchase extendable classes such as $C$ from a software vendor, or when $C$ is from a class library. In this chapter we will show that standard black-box specifications are not sufficient to do such reasoning, but that specifications that provide information about object interactions via method calls are sufficient. Such specifications are often referred to as grey-box specifications [11, 16], and we will use the same terminology. In the next chapter, we will show how such grey-box specifications can be checked during runtime, even in the absence of the source code under test.
Figure 2.1: Order class code

2.2 A Simple Example

To illustrate the need for grey-box specifications, we will examine a variation of a simple example from [10]. Here, we have an online mail order company that sells items, represented by objects of type Item, where customers can purchase multiple items by adding them to a single Order. As part of the online system, customers are also allowed to combine unshipped orders together into a single order. The code for Order can be seen in Fig. 2.1. The data member items holds onto the items in the order using a Java Vector, and totalPrice keeps track of the total price (in cents) of all of the order items. The method addItem() is used to add a single item to the order,
and \texttt{combine()} combines orders by adding the contents of the \texttt{o2} order parameter into the current (i.e., \texttt{this}) order.

Here, \texttt{combine()} is implemented so that it calls \texttt{addItem()} repeatedly on each of the items to be added from the order \texttt{o2}. \texttt{combine()} is acting as a template method, whereas \texttt{addItem()} is its only hook method. By designing our system in this way, developers who wish to create derived classes of \texttt{Order} with enhanced features only need to redefine the \texttt{addItem()} method—this is the main motivation behind the template method design pattern [33]. Polymorphism guarantees that \texttt{combine()}, when called on these enhanced \texttt{Order} objects, will invoke the new definition provided for \texttt{addItem()}. By using these new definitions, the behavior of \texttt{combine()} itself is enhanced, even though its own definition has not been changed. Leveraging polymorphism in this way allows us to handle adding items to an order, whether by adding individually or through order combination, in a uniform way without having to rewrite every method.

To further enforce this way of programming, where we do not redefine each method, the Java keyword \textit{final} is used in the definition for \texttt{combine()}. The \textit{final} keyword prevents the redefinition of a method in a derived class.

Now, suppose that a separate group of developers want to slightly enhance the \texttt{Order} class by allowing “bonus points” to be awarded for each item purchased that has a price of $25 or over. These points could possibly be accumulated and later redeemed by the customer for discounts on purchases, although such details are outside of the specific part of the system we are examining. To get a version of the \texttt{Order} class with the addition of bonus points, we can extend \texttt{Order} as shown in Fig. 2.2.

Class \texttt{BPOrder} inherits the \texttt{items} and \texttt{totalPrice} data members from \texttt{Order}, as well as the method definition for \texttt{combine()}. To implement handling of bonus points,
public class BPOrder extends Order {
    protected int bonusPts;
    public BPOrder() { bonusPts = 0; }
    public void addItem(Item x) {
        if (x.getPrice() >= 2500) bonusPts++;
        super.addItem(x);
    }
}

Figure 2.2: BPOrder class code

BPOrder only has to introduce the bonusPts data member, along with a constructor, and a redefinition of addItem() that increments bonusPts as necessary. Due to polymorphism, the inherited definition for the template method combine() will call the addItem() hook method defined in the new BPOrder class when invoked on BPOrder objects, and will therefore update bonusPts as needed when called.

2.3 Reasoning with Black-box Specifications

2.3.1 Abstract and Concrete Specifications

Before we examine the specifications for our example, we should first say a few words about the use of concrete versus abstract specifications. The OO principle of abstraction prescribes that a client of a class C should not be concerned with how its operations manipulate its concrete state—the state of the data fields of the class [13, 40]. A standard approach [45, 74] is to specify the behavior of the methods of a class is in terms of an abstract model of C, rather than its concrete state. In our discussion, however, we are not taking the client view, but rather the view of a derived class designer who wants to reason about the behavior of his or her derived class D of C. Because the methods of D can access the non-private variables of C, the derived
class designer may necessarily have to work with the concrete model of C. For this reason, we will be using concrete specifications during our discussion in this chapter and the next.

We should note that there are alternative approaches, such as RESOLVE [83], that advocate a high degree of component encapsulation, where derived classes are only allowed to interact with parent classes through their public interfaces. Such approaches, however, do not allow implementers to utilize the full power of OO by strongly limiting how polymorphism and dynamic dispatch can be leveraged. Another approach, a sort of specification “middle-ground”, can be found with the Java Modeling Language (JML) [54]. JML provides abstract models that can be used in specifications, while also allowing for concrete variables to have a higher level of visibility in the specifications. For example, a private variable of C can be made to be “spec-protected” so that derived classes such as D can refer to it in their specifications without violating abstraction constraints.

This brings up the issue of field visibility. Since we are mainly concerned with reasoning at the derived class level, in our examples we generally define the fields of base classes as having protected visibility. Private fields would not be allowed to appear in the specifications provided to derived class developers, and would not (directly) take part in reasoning about the derived class behavior. In practice, information about private variables that would be needed for reasoning here would be provided in an abstract model. Since our focus is not on abstract modeling, and we usually have to work with the concrete state of C in derived classes regardless, we use protected fields to allow for us to specify things in a completely concrete manner.
pre.Order.new() ≡ true
post.Order.new() ≡ this.totalPrice = 0 ∧ this.items = ⟨ ⟩

pre.Order.addItem(x) ≡ true
post.Order.addItem(x) ≡ this.totalPrice = #this.totalPrice + x.price ∧ this.items = #this.items * ⟨ x ⟩

pre.Order.combine(o2) ≡ true
post.Order.combine(o2) ≡ this.totalPrice = #this.totalPrice + #o2.totalPrice ∧ this.items = #this.items * #o2.items

Figure 2.3: Black-box specifications for Order

2.3.2 Specifying the Order Class

Now, let us look at the black-box specifications for Order in Fig. 2.3. Here, we use the prefix ‘#’ in front of a variable to denote its value at the beginning of the method call. In the specifications, Java Vectors are modeled by sequences of elements, and ‘*’ is the concatenation operation for sequences.

Now, suppose the developers of the BPOrder class want to determine exactly what behavior its methods will exhibit, given that they have the black-box specifications of Order as shown in Fig. 2.3, as well as their own code in Fig. 2.2. They do not, however, have the source code of Order. This is often the case when one party purchases code from another, and source code is not provided for proprietary reasons. Also, in the case where pre-compiled library classes are used, source code may not be readily available for examination.

In this situation, it is possible to arrive at the behavior for BPOrder.addItem(). From the code in Fig. 2.2, we see that bonusPts will be incremented by one if x.price is at least 2500, and that items and totalPrice will be updated according to its base
class specification, due to its call to `super.addItem()`. In a similar way, we arrive at the behavior of the constructor `BPOrder()`, where `bonusPts` and `totalPrice` are 0, while `items` is the empty vector.

But what about the behavior of the template method `combine()` when invoked on a `BPOrder` object? From the code for `Order` in Fig. 2.1, we know that `combine()` makes calls to `addItem()`, and so we would know what effects it would have on `bonusPts` as well as `items` and `totalPrice`. However, since we are not allowed to look at the source code for `Order`, but only its black-box specifications, we would not have this crucial information.

We can note that the redefinition of `addItem()` in the derived class, as well as the new constructor, both satisfy their corresponding base class specifications found in Fig. 2.3. In effect, we can conclude that `BPOrder` is a behavioral subtype of `Order`. A class `D` is a behavioral subtype of a class `C` that it extends, if all of `D`’s methods found in `C` adhere to the specifications given for them in `C` [56]. We can conclude that the inherited `combine()` will still behave as specified in Fig. 2.3 using the following argument: When reasoning about `combine()`’s behavior in the base class, we would have appealed to the base class specification for `addItem()`, if it indeed was called. Since the redefined `addItem()` still adheres to the original base class specification, the same reasoning could be used when determining the behavior of `combine()` when applied to a `BPOrder` object, and so it would still adhere to the base class specification. In other words, we can conclude that `combine()` updates `items` and `totalPrice` as before.

What behavioral subtyping gives us, in general, is a guarantee that methods inherited by the derived classes will adhere to their base class specifications, given that the redefined methods adhere to their base class specifications. For this reason, we
can predict what effects the methods of a derived class will have on a derived class object’s projected base class state. Those methods not newly introduced in the derived class will have the same effect on the base class state as their corresponding methods in the base class; newly introduced methods can be reasoned about directly from their code, and using the specifications of the other methods it may call. The difficulty arises when we try to reason about their effects on the extended state introduced in the derived class.

In our example, this extended state comes from the bonusPts field. Although we can say what effect combine() has on its base class state, we do not know what it will do to bonusPts on the basis of the specification in Fig. 2.3. We can see this by considering an alternative implementation for Order.combine():

```java
public final void combine(Order o2) {
    items.addAll(o2.items);
    totalPrice += o2.totalPrice;
}
```

This implementation also adheres to the specification given in Fig. 2.3, but does so through directly manipulating the data members instead of calling addItem(). When initially coding Order, if we did not consider the possibility of enhancing it in the future via extension, such an implementation may seem more reasonable than the original due to efficiency concerns. Now, suppose we used this implementation in Order, and BPOrder was created by extending Order as before. The behavior of combine() when invoked on such a BPOrder object would be different from before; specifically, bonusPts would never be incremented when combine() is called, even if some of the items in the order being added are over $25 in price. With the black-box
specifications of \texttt{Order}, but not its source code, we would not be able to predict what \texttt{combine()} would do to \texttt{bonusPts}.

### 2.4 Trace-based Specifications

In order to reason about the extended state and the richer behavior of \texttt{BPOrder}, what are needed are specifications that provide more than the standard black-box information. One solution is to provide \textit{trace-based specifications}, which outline which (alterable) methods will be called on what parameters, what will be the sequence of calls, etc. To represent the sequence of calls made, we introduce the \textit{trace} variable $\tau$, which is an \textit{auxiliary} variable [64] which stores the information about the hook method calls $t()$ makes during its execution. When the method $t()$ starts execution, $\tau$ will be the empty sequence, since at the start $t()$ has not made any such calls. As $t()$ executes, information about each hook method call it makes will be recorded on $\tau$. In our postcondition specifications for $t()$, we will not only have information about the state of the object in question when $t()$ terminates, but also about the value of $\tau$, i.e., about the hook method calls $t()$ made during its execution. It should be stressed that $\tau$ is not a program variable, but a variable introduced only for specification purposes to represent the call sequence. However, in conceptualizing $\tau$ and its contents, it can be useful to think of it as if it were an actual program variable.$^3$

A preliminary trace-based specification for \texttt{Order} is shown in Fig. 2.4. The specifications for the constructor and \texttt{addItem()} are the same as those found in the standard

\footnote{In later chapters dealing with monitoring, $\tau$ will be introduced as a program variable as part of the testing frameworks.}
Figure 2.4: Trace-based specifications for Order (preliminary)

black-box specifications shown in Fig. 2.3, since they do not make calls to other methods. Assuming that derived classes of Order will be proper behavioral subtypes, we can also assume that combine() will update totalPrice and items as before. However, because combine does make calls to the enrichable method addItem(), we must also include this information in its postcondition. In the notation, for each hook method call made by a template method t(), one record is stored in τ in the order in which it was called. These record elements in τ are indexed (starting with index 0) as would be elements of an array. We can access the name of the hook method called for a particular record using the dot notation along with the specification keyword method.

We can also access the arguments passed to the hook method with the keywords arg1, arg2, . . ., argk. The postcondition for combine() states, in addition to the information provided in its standard black-box specification, that |o2| hook method calls were made, and each of them were to addItem(), and the parameter passed for the i + 1st call was the i + 1st element of o2. Note that we use the “==” operator here to
represent reference equality to distinguish it from the “=” operator, which stands for value equality in our notation.

This trace-based specification gives us an understanding of what `combine()` does, and through informal reasoning using this specification and the code for `BPOrder`, we might come to the conclusion that it will update `bonusPts` properly when invoked. However, the specifications in Fig. 2.4 are not quite complete, and do not disallow certain unwanted behaviors. For example, when `combine()` is invoked, it could possibly first directly add the prices of each of the items in `o2` to `totalPrice` without calling `addItem`, then temporarily change all of these items’ prices to zero, and then proceed to invoke `addItem` on each of `o2`’s items. This would ensure that `bonusPts` would not be changed. Although this particular situation is somewhat artificial and contrived, the same type of problem can arise more naturally in more complex settings, and we need to be sure that our specification formalism is capable of appropriately addressing such cases.

Roughly speaking, what is needed is additional information about what state changes are or are not made by the template method’s code during execution. By simply stating what hooks a template method t() calls at what times, we are leaving out information about what changes t() may make to the state directly or by invoking non-hook methods.\(^4\) Thus, information must be provided about what happens during the execution of t():

\(^4\)One possible alternative is to define the semantics of trace-based postconditions such that t() is assumed not to make changes to state directly, unless such changes are stated explicitly in the specification. Because our eventual goal is to evaluate trace-based specifications during runtime monitoring, it would be necessary to infer when the state is not supposed to change from the specifications, and generate the proper runtime checking code to do this check (assuming we desire to check the complete specification). To simplify the task of generating monitoring code, and also to give the tester more freedom to decide exactly what conditions to check, and what to leave out, this alternative definition was not adopted.
1. Before any hook methods are called,

2. Between the hook method calls, and

3. After all the hooks are called until the method is finished executing.

This can be illustrated using Fig. 2.5, where the template method $t()$ makes calls to hook methods $h1()$, $h2()$, and $h3()$. Here, item 1 is represented by the interval between $p$ and $a_1$, item 2 by the intervals $b_1$ to $a_2$ and $b_2$ to $a_3$, and item 3 by the interval $b_3$ to $q$. The change in state (or the lack of state change) occurring in each of these intervals can be specified by a relation on state, using the records of $\tau$ and the pre- and post-states of $t()$ to reference the necessary states at the interval endpoints.

From this diagram, we can also gain some intuition as to why trace information, as well as information internal to $t()$, is needed to fully be able to determine the final behavior of $t()$. Because hook methods’ behaviors can be enriched or changed, they must be viewed as something external and changeable in relation to $t()$. However, with the trace variable $\tau$, we do not have to have full knowledge about what happens at these “hot spots” where the hooks are called—we can use a trace record at each such
spot to record the pre- and post-states\textsuperscript{5} for that particular hook call. Using these trace records, we can relate the hook states within the context of the body of \texttt{t()}, to arrive at a specification for \texttt{t()} which is essentially parameterized by the records on \( \tau \). When we want to arrive at \texttt{t()}’s behavior resulting from the use of specific implementations of the hook methods, we need only to “plug-in” the relations between the beginning and ending states for the particular hook methods into the trace-based specification for \texttt{t()}. A hook method that does not call enrichable methods can be specified with standard black-box specifications. Such specifications give us the input/output state relation in a direct way. However, hook methods that can call enrichable methods are template methods themselves, and thus also will have traces in their specifications. This can make reasoning more challenging if some template methods are recursively related, but can be handled in much the same way as reasoning about recursion in procedural programs. A formal treatment of reasoning with traces can be found in [79, 77, 31], where proof rules are given and are shown to be sound and relatively complete with respect to first-order logic.

Although the Order example that we have been examining consists of a single class where the template method and its (original) hook method reside, it should be emphasized that this discussion applies to more complex multi-class examples. We will turn to such an example in the next chapter, and further discuss specification as well as testing issues there. To generalize, a \textit{template method} of a class \texttt{C} is a method whose behavior can be changed and/or enriched through redefining methods that it calls; these redefinable methods are designated as its \textit{hook methods}. These hook methods can reside in classes other than \texttt{C}. For example, suppose \texttt{C} has a data

\textsuperscript{5}Or more precisely, give us a means to refer to these states in the specifications.
member d of class D, and it is possible that the object bound to d can be changed at runtime, and the object can be of a different class than as before. Now, if a method t() of C invokes a method m() on d in its body (without changing the object bound to d beforehand), and m() is a redefinable method of D, then m() is a hook of t(). Suppose E is a derived class of D, and m() is redefined in E. If the object bound to d is of class E, then when t() makes the call d.m(), it can have a different net effect than if d was of class D.

To determine if a method m() is a hook of t(), it has to be possible for t() to invoke (either directly, or indirectly via another method) different definitions of m() via dynamic dispatch, including those that may be provided in future derived classes. Otherwise, m() does not provide a point of change for t(). This idea of determining method dependencies in the face of polymorphism is not new, and has been discussed in the literature [38, 48, 72]. Our focus is on determining whether a method m() is a hook of a given method t(), however, and not about call dependency graphs. We will note, though, that the hook-template relations for a system can be used to generate such graphs.

In order to modularly reason about the wide variety of possible extendable systems, trace records must contain information about what method was called, on which target object it was called, what state the target and parameters were in before and after the call, and for functions, the return value [77]. Here is the complete list of the pieces of information stored for each hook call record, along with the notation used to refer to them:

1. The name of the hook method invoked, with parameter types to resolve overloaded method names: method
2. The reference to the object on which the hook method was invoked: \textit{target}

3. The state of the \textit{target} object immediately before the hook call: \textit{pre.target}

4. The \textit{target} object’s state immediately after the hook call: \textit{post.target}

5. The references to the parameters passed to the hook method\textsuperscript{6}:

\[ arg_1, arg_2, \ldots, arg_k \]

6. The parameters’ state before the hook method call:

\[ \text{pre.arg}_1, \text{pre.arg}_2, \ldots, \text{pre.arg}_k \]

7. The parameters’ state after the hook method call:

\[ \text{post.arg}_1, \text{post.arg}_2, \ldots, \text{post.arg}_k \]

8. The reference to the object, or value, returned (if any) by the hook method:

\[ \text{result} \]

9. The state of the object returned (if any) by the hook method: \textit{post.result}

In specifications, we need to explicitly make a distinction between an object’s reference and state when dealing with a hook call’s target, arguments, or return object. If object references alone were saved in the trace, and not the states, we would only be able to access the current state at the end of \texttt{t()}’s execution via the references, but not the intermediate states when the hooks are called. In specifications, to help differentiate between value equality and reference equality, we use the `\texttt{=}' and `\texttt{==}' operators, respectively.

\textsuperscript{6}Since we are using Java semantics, where parameters are passed by value-reference, we do not need to distinguish between the parameter references before and after the call. The reference values that are passed as arguments cannot be changed by the called method.
As was mentioned earlier, the postcondition for `combine()` in Fig. 2.4 was not complete in the sense that it did not prevent certain pathological behaviors. Specifically, no restrictions were made on what state changes `combine()` could make before, between, and after the hook method calls, as long as the final state of `totalPrice` and `items` were what they should be. To “tighten” up the postcondition, we can put in additional requirements relating these intermediate states to one another; such an alternative is shown in Fig. 2.6. Note that we do not directly mention `this.totalPrice` or `this.items` here, as we did in Fig. 2.4. We will see shortly that this information can already be inferred from these specifications, assuming we have behavioral subtyping.

The first line of the postcondition simply states that the number of hook calls made is the same as the number of items in `o2`. The start of the quantification on the next line iterates through each of the indexed trace records, and first states that each hook call is made to `addItem()`. The next line states that for hook method invocation `i`, the target is `this` and the `i`th element of `o2` is the single parameter argument. In the next line, the specification says that the states of the elements of `o2` do not change
from the start of the invocation of combine() to the time each of the hook methods are called. The next line says that in between each of the hook method calls, the this object’s state is not changed. The next two lines, which appear outside of the scope of the quantification, state that the state of this is also unchanged from the beginning of execution of combine() to the first hook call, and also after control returns from the last hook call to the end of combine()’s execution. The first part of the very last line tells us that the vector of items of o2 is unchanged, or more precisely, the vector of references is unchanged. This allows for the individual Item objects of o2.items to change when the hook addItem() is invoked. The final part of the last line states that the totalPrice of the Order o2 is not changed by combine(). Because the hook addItem() is not given a reference to the whole order o2, and since Items do not have a reference to their enclosing Orders, the hook calls cannot cause combine() to indirectly modify totalPrice during execution.

With such a specification, we can see exactly what combine() does inside of its body to the object on which it was invoked, and to the Order parameter o2. Assuming we have behavioral subtyping, we can make conclusions such as \( \text{totalPrice} = \#\text{totalPrice} + \#o2.\text{totalPrice} \) by plugging-in the information from addItem()’s specifications, since we know that addItem() will update totalPrice in the expected way if it is redefined in a derived class. In cases where we do not assume behavioral subtyping, specifications such as those in Fig. 2.6 are very useful because they minimize assumptions made about the hook methods they call while being precise enough to allow for us to infer the resulting behaviors from plugging-in these hook methods whose behavior we did not predict beforehand.
2.5 Related Approaches

The approach to grey-box specification we presented here is based on the work of Soundarajan and Fridella [77, 78], where the behavior of t() in the base class is specified using traces that record its calls to hook methods. This work also shows how such specifications can be used to formally arrive at the richer behavior of t() by combining it with the derived class specifications for the hooks. Helm, Holland, and Gangopadhyay [39] introduce the notion of contracts to specify obligations among cooperating objects, which can include information about what calls to certain hook methods must do, and under what conditions they will be called. These contracts allow for refinement when parts of the system are specialized. Buchi and Weck [16] discuss the need for specifications that contain information about hook method calls, and focus on developing special notations to simplify specifications, such as model programs with the hook calls interspersed among formal assertions.

Froehlich, Hoover, Liu, and Sorenson [32] generalize the idea of hook to denote any aspect of the system—not just single hook methods—that can be used to enrich or specialize the system. They introduce a syntactic notation for these generalized hooks but do not consider specification of behavior or testing. Kirani and Tsai [50] discuss the importance of specifying the sequence of calls a method t() makes during its execution, as well as testing against this specification. However, they are interested in all calls, not just hook method calls. While the sequence of all calls t() makes is useful in understanding how the body of t() works, the calls that a template method makes to non-hook methods are inessential for modular reasoning, since the effect of these calls on t() cannot be changed through leveraging polymorphism.
Ruby and Leavens [72], building on earlier work by Kiczales and Lamping [48, 51], present a formalism where some additional information about a method beyond its functional behavior is provided. This may include things such as information about the variables the given method accesses, and what hook methods it invokes. While this is not as complete as the information that can be provided using traces, it has an advantage that it is relatively easy to build tools that can mechanically extract this type of information from the code and exploit it, rather than having to be specified by the designer.
CHAPTER 3

MONITORING GREY-BOX SPECIFICATIONS

In the previous chapter, we saw that many of the important behaviors exhibited by object-oriented systems cannot be expressed using standard black-box specifications, but require some form of grey-box information. We showed how such behaviors can be specified using trace variables, and how trace-based specifications allow us to reason incrementally. In this chapter, we will continue this discussion using a more complex multi-class example from the area of OO frameworks. Using such a framework as a running case study, we will then turn to how we can monitor such specifications during runtime, even when the source code of the class methods being checked is not available.

3.1 Problem Overview

The task of monitoring trace-based specifications is challenging because we need to keep track of the value of the trace $\tau$ during runtime. Every time $t()$ makes a call to $h()$ (or any other hook method), $\tau$ has to be updated to record information about

$^7$By monitoring, we mean collecting information at runtime to evaluate assertions made in the specifications, and alerting the user when they are violated. This is different from what we will refer to as testing, which also entails the selection of test cases to be run. Here, monitoring techniques facilitate testing, although they do not say anything about what test cases to use. We will discuss test case selection in the final chapter.
this call and its subsequent return. However, unless there is monitoring code built into \( t() \) that tracks such calls, such information cannot normally be extracted during runtime. It is entirely unlikely that the original code body of \( t() \) would include the code necessary to update \( \tau \), since \( \tau \) is a variable introduced as part of the specification language in order to help reason about the behavior of \( t() \); it is not a variable that was included in C’s code by its designer.

One possible solution to this problem would be to introduce the variable \( \tau \) to the system, and modify the body of \( t() \) to include suitable statements that would update it during execution. Such statements would include assignments and calls to methods that allowed for copying object state, and they could be inserted immediately before and after each hook method call, However, such an approach does modify the code being examined and monitored, and can cause problems if not done correctly. Furthermore, such modifications are not even possible if we do not have access to the body of \( t() \) in C. Another approach—one that does not rely on source code, but rather, exploits polymorphism in the same way that template methods do—is presented in this chapter, along with a tool that automates the creation of such monitoring code.

### 3.2 Object-Oriented Frameworks

A well-designed object-oriented framework [15, 44] for a given application area can serve as the key component for applications built on it [21]. An early example was the MacApp framework [3] that provided many of the common functionalities of applications for the Macintosh, thereby reducing the work involved in building a new application, and ensuring a uniform look-and-feel among the applications. Frameworks provide prime examples of real OO systems that leverage polymorphism,
inheritance, and dynamic binding—the features of OO that can complicate the tasks of reasoning about and specifying such systems. For this reason, we will continue our discussion on specifying OO systems using an OO framework as a case study, and turn to how such specifications can be monitored later in the chapter.

An object-oriented framework is a class, or collection of classes, that implements basic functionalities that are common across several different application domains. The framework component itself should only provide general behaviors; it is the application developers who are responsible for specializing these frameworks to meet their particular needs. The framework contains one or more template methods that provide these general behaviors, which often entails mediating the interactions between different objects in the system. The application developers, who generally are different from the framework developers, specialize the behavior of these frameworks by providing the appropriate code for the hook methods of the framework. These hook methods are intended to be specialized to reflect the needs of the specific application. By employing objects of the derived classes containing the redefined hooks in the final framework application, calls made by the template methods to the hook methods will be dispatched at runtime, via polymorphism, to their definitions in the derived classes. This ensures that the template methods defined in the framework exhibit behaviors customized to the needs of the application, even though the template methods' code has not been modified.

Because the interaction patterns implemented in the template methods are often the most involved aspect of the total behavior required in the application, using framework components can considerably reduce the amount of effort required for developing a new application. With the Order example from the previous chapter, we
have already seen the need for additional information beyond what is contained in standard black-box specifications. With frameworks, this need is much more acute, because frameworks are primarily characterized by their interaction patterns. Indeed, the point of frameworks would be completely lost if we only used black-box specifications.

3.2.1 An Example: Diagram Editor

“Node-and-edge” diagrams are common in a number of domains. Some examples are road maps where the nodes are cities and edges, perhaps of varying thicknesses, represent highways; electrical circuit diagrams, where the nodes represent such devices as transistors, diodes, etc., and the edges represent wires and other types of connections between them; control flowcharts, where the nodes represent different statements of a program, and the edges represent the possibility of control flowing from one node to another during execution. In each of these domains, a diagram editor that allows us to create and edit diagrams consisting of the appropriate types of nodes and edges is obviously very useful. While each of these diagram editors can be created from scratch, that is clearly wasteful since these diagrams, and hence also the diagram editors, have much in common with each other.

A much better approach is to build a framework component that contains all the common aspects, such as maintaining the collection of nodes and edges currently in the diagram, tracking mouse movements, identifying, based on mouse/keyboard input, the next operation to be performed, and then invoking the appropriate (hook-method) operation provided by the appropriate node or edge class. A developer interested in building a diagram editor for one of these domains would then only have
to provide the derived classes for Node and Edge, appropriate to the particular domain. Thus, for example, to build a circuit diagram editor, we might define a TransistorNode class, a DiodeNode class, a SimpleWireEdge class, etc. Once this is done, the behavior of the template methods in the framework will become customized to editing circuit diagrams, since the calls in these methods will be dispatched to the definitions in the classes DiodeNode, SimpleWireEdge, etc.

For the remainder of this chapter, and throughout the next, we will use a simple diagram editor framework, modeled on the one in [40], as a running case study. Fig. 3.1 shows a diagram editor built on this framework component, in use. In this application, there are two Node (sub)types (Circle and Square) and two Edge (sub)types (SolidLine and DottedLine). The application window is split into two parts. The left part consists of a collection of buttons, the first three of which correspond
to basic functions ("Exit", "Move", and "Erase"), the next two of which allow us to draw two different Node objects ("Circle"s and "Square"s), and the last two for drawing each possible Edge type ("Solid Line"s and "Dotted Line"s). The right part of the window displays the current diagram. The user can click on one of the Node buttons in the left part, causing the button to be highlighted, and then if the user next clicks anywhere on the right part of the canvas, a new Node object of that particular type will be created and placed at that point. The Edge buttons are similar, except that after clicking the Edge button, the user must click one Node object and drag to another Node to place an Edge of that type, connecting the two Nodes. Clicking on the "Move" will cause that button to be highlighted; if the user next clicks on a Node object in the current drawing and drags it, the Node will be moved to its new location. Furthermore, any Edges incident on that Node will be redrawn appropriately. Edges cannot be moved on their own. Clicking the "Erase" button will highlight it, and if the user next clicks on a Node object in the drawing, that Node and all its incident Edges will be erased; clicking on an Edge will erase that Edge. Clicking on "Exit" will, of course, terminate the diagram editor.

Figure 3.2 contains the Unified Modeling Language (UML) [14] model of the framework, as well as an application built upon it. The functionality corresponding to tracking mouse movements and clicks, determining which action or Node or Edge type has been selected, etc., is provided in the DiagEditor class. Our focus will not be on mouse-driven events, but rather on how the framework manages the Nodes and Edges, carries out the user commands, and interacts with the graphics Scene. To achieve the actual display of the individual Nodes or Edges, the method setUp() should first be invoked to initialize the framework and its display. Then, as the user
Figure 3.2: Diagram editor framework
issues different commands via mouse-driven events during execution, the appropriate
handler methods—`addNode()`, `addEdge()`, `moveNode()`, `eraseNode()`, and `eraseEdge()`—
are invoked. These handler methods are template methods; they do their jobs by
calling the appropriate hook methods defined in the appropriate derived classes of
`Node` and `Edge`. For example, when `addNode()` is called, the appropriate `draw()` hook
method needs to be called in one of `Node`’s subclasses, since different types of `Nodes`
will most likely be rendered differently in the display. Indeed, this is the key aspect
that distinguishes the diagram editor for a given domain from that for another domain.

“Hit-testing”, i.e., testing whether the current pointer location lies on a given `Node`
or `Edge` object, is also something that has to be determined by hook methods defined
in the appropriate derived classes, since the shape and sizes of these objects is not
known to the framework component.

One important practical question that the framework has to deal with is how it
will maintain the specific kinds of `Nodes` and `Edges` and their corresponding buttons.
The framework itself does not know what the derived `Node` or `Edge` types are, nor
even how many such types will be used in any specific application, since all this will be
decided by the application developer when he or she customizes it for the particular
application domain by providing the corresponding derived classes. This situation is
handled by requiring that at its start, the diagram editor application must “register”
each type of `Node` and `Edge` specific to the application by calling `registerNode()` or
`registerEdge()`, using an instance of each of these types as its parameter. The code that
makes these calls to the register methods should reside in the `registerAll()` method,
which needs to be redefined in a derived class of `DiagEditor`. To set up the diagram
editor window, the user will invoke `setUp()` on an object of this derived class (`setUp()`
itself will be inherited from :ref:`DiagEditor`. It is the :meth:`setUp()` method that invokes the redefined version of :meth:`registerAll()`, as well as initializes the editor window and drawing area. These registered instances are represented by the :code:`n_types` and :code:`e_types` roles pictured in Fig. 3.2. These instances also serve as the prototypes for cloning when the user chooses to add a :obj:`Node` or :obj:`Edge` of a given type to the diagram.

Since our goal is to focus on behavioral and monitoring issues, we have left out the details of the user’s interactions with the framework via the mouse. Instead, we will treat the handler methods as the point of user interaction with the system, rather than having to analyze mouse movements. The :obj:`Scene` in Fig. 3.2 represents the “graphics scene”, which in practice would be a graphics window, the screen buffer, or something else of this nature. In any case, the :obj:`Scene` contains information that correlates to an appropriate description of what is displayed by the graphics editor, such as shown in Fig. 3.1. Since our interest is not in how the graphics themselves are rendered, we will not go into details of how the :meth:`draw()` and :meth:`unDraw()` methods interact with the :obj:`Scene`.

In the next section we consider segments of the code of some of the methods of the framework, and focus on the handler template methods. We will first examine how can we specify their behaviors using traces so that an application developer can “plug-in” their redefined hook methods’ specifications to arrive at the corresponding customized behavior. Later, in Sec. 3.3, we will show how such specifications can be monitored during runtime.
3.2.2 Diagram Editor Implementation

Figure 3.3 contains portions of the Node class implementation as it may be defined in the framework. A basic Node contains a single field, its center point, which acts as

```java
class Node {
    protected Point center;
    Node() { center = new Point(); }
    Node(Point p) { center = p; }
    protected String className() { return "Node"; }
    protected void draw(Scene sc) {... render a generic Node...}
    protected void unDraw(Scene sc) {... erase this Node from the Scene...}
    protected Object clone() { return new Node(center); }
    protected boolean isOn(Point p) {... is p on the Node? ...}
    protected Point boundPt(Point p) {... return a boundary Point of the Node...}
    final protected void move(Point p) { center = p; }
    final protected Point center() { return center; }
}
```

Figure 3.3: Node class code

its anchor or reference point in the drawing Scene. A default constructor is provided, along with another that sets its center. The method className() is defined such that it returns the name of the class, and is meant to be redefined in derived classes of Node to reflect the specific class. The diagram editor uses className() to provide proper labels for the buttons when it is initialized at the time DiagEditor.setUp() is called. The draw() and unDraw() methods are used to render and eliminate the given Node from the Scene; these methods should also be redefined in the specialized Node classes. The method clone() provides a copy of the given Node, and will be invoked on the registered Node prototypes when such a copy is needed to be added to the
The `clone()` method must return a Java `Object` because it overrides the `clone()` method from `java.lang.Object`, which itself returns an `Object`. The method `isOn()` is used for hit-testing in conjunction with mouse clicks, to determine if the `Node` in the `Scene` has been selected. The method `boundPt()` returns the point on the boundary of the graphical representation of `Node` that is closest to the point `p`. This method is intended to be used by `Edge.draw()` to locate endpoints in the `Scene` between which it is safe to draw the `Edge`, so that the rendered `Edge` does not overlap the rendered adjacent `Nodes`. The methods `clone()`, `isOn()` and `boundPt()` should also be appropriately redefined in the derived classes. The last two methods, `move()` and `center()`, are defined as `final`, since their implementations are independent of the specifics of a particular type of `Node`, and thus should not be redefined. `move()` changes the location of the `Node`'s center, but does not play any part in re-rendering the `Node` in the `Scene`. The method `center()` simply returns the `Node`'s center point.

In Fig. 3.4, we show an implementation for the `Edge` class of the framework. The `Edge` class is very similar to `Node`, except that instead of keeping track of a center point, an `Edge` object has references to the “from” and “to” `Nodes` that it joins. `Edges` are anchored by the location of their adjacent `Nodes`, and will be moved accordingly whenever these `Nodes` are moved; no central anchor point for `Edges` are needed. The methods of the `Edge` class are analogous to those with the same names found in the `Node` class. The “getter” methods `from()` and `to()`, and the “setter” method `connect()`, do not have to be redefined in derived `Edge` classes, and are therefore defined as being `final`.

It may be worth noting that in an actual framework of this kind, it might be more appropriate to have these classes as `abstract` in the framework, with appropriate
class Edge {
    protected Node from, to;
    Edge() { from = new Node(); to = new Node(); }
    Edge(Node f, Node t) { from = f; to = t; }
    protected String className() { return "Edge"; }
    protected void draw(Scene sc) { ...render a generic Edge... }
    protected void unDraw(Scene sc) { ...erase this Edge from the Scene... }
    protected Object clone() { return new Edge(from, to); }
    protected boolean isOn(Point p) { ...is p on the Edge? ... }
    final protected Node from() { return from; }
    final protected Node to() { return to; }
    final protected void connect(Node f, Node t) { from = f; to = t; }
}

Figure 3.4: Edge class code

implementations in the applications. However, as we will later see, this decision—whether the classes to be specified and monitored are abstract or concrete—will not affect the overall methodology that is used.

Code for the diagram editor class\textsuperscript{8}, DiagEditor, is shown in Figs. 3.5, 3.6, and 3.7. Starting with Fig. 3.5, we show four data members of type Vector (which we use as our principal collection type), nodes, edges, n_types, and e_types. The data members nodes and edges simply keep track of the nodes and edges currently in the diagram, and n_types and e_types hold onto the prototype objects used to represent the node and edge types as previously described. The Scene sc is the object containing the graphical representation of the diagram. The method addNode() is responsible for adding a node of a specific type to the diagram by first making a clone of the nd+1st

\textsuperscript{8}Exception handling code for calls to Vector.get() with invalid indices is not needed, since this is an unchecked exception in Java.
class DiagEditor {
    protected Vector nodes, edges, n_types, e_types;
    protected Scene sc;

    final protected void addNode(int nt, Point p) {
        Node n = (Node)(((Node)n_types.get(nt)).clone());
        n.move(p); nodes.add(n); n.draw(sc);
    }

    final protected void addEdge(int et, int fn, int tn) {
        Edge e = (Edge)(((Edge)e_types.get(et)).clone());
        e.connect((Node)(nodes.get(fn)), (Node)(nodes.get(tn)));
        edges.add(e); e.draw(sc);
    }

    final protected void moveNode(int nd, Point p) {
        Node n = (Node)(nodes.get(nd));
        n.unDraw(sc);
        for(Iterator iter = edges.iterator(); iter.hasNext(); ) {
            Edge e = (Edge)(iter.next());
            if (e.from()==n || e.to()==n) e.unDraw(sc);
        }
        n.move(p); n.draw(sc);
        for(Iterator iter = edges.iterator(); iter.hasNext(); ) {
            Edge e = (Edge)(iter.next());
            if (e.from()==n || e.to()==n) e.draw(sc);
        }
    }

    ...}

    Figure 3.5: DiagEditor class code (part 1)

node in the n_types vector, setting its center point to p, inserting it into nodes, and then renders it in the Scene by calling on the draw() hook method (re)defined in the proper Node or Edge class. It should be noted that addNode() is not responsible for checking what button is currently selected, tracking mouse clicks to determine the center point for the Node, etc. These tasks should be done by other methods of the DiagEditor class (not pictured in the UML diagram in Fig. 3.2, or shown in the code) that handle interfacing with the user, and will be the ones that call methods such
as `addNode()`.

The method `addEdge()`, similar to `addNode()`, creates a clone of the `ed+1st` edge type prototype, connects the edge to nodes `fn` and `tn` in the `nodes` vector, adds it to `edges`, and finally renders it.

The method `moveNode()` is responsible for moving the `nd+1st` node of `nodes` to point `p`, which should not only change the center of that node, but should also be reflected in the graphical `Scene` itself. This process should not only re-render the node, but also all of its adjacent edges. To do this, first the node is removed from the `Scene` via `unDraw()`, along with its adjacent edges. In the code, the `edges` are iterated through, and checked for adjacency to the `Node` `n` using the `from()` and `to()` methods of `Edge`. Then, the center of the node is moved via `move()`. Lastly, the node and all of its adjacent edges are re-rendered. An alternative implementation for `moveNode()` would be to simply change the node’s center using `move()`, and then call the `refreshScene()` method (shown in Fig. 3.6) to clear and then re-render everything in the diagram.

In Fig. 3.6, we start with the method `eraseNode()`, which eliminates a given node and all of its adjacent edges from the diagram and the associated `Scene`. First, the `nd+1st` node of `nodes`, `n`, is erased from the screen and removed from `nodes`. Then, we iterate through all of the `edges`, and for those adjacent to `n`, we erase those also from the `Scene` and remove them from `edges`. The next method, `eraseEdge()` is simpler, since only the specified edge needs to be removed from `edges` and from the `Scene`, and no nodes of the diagram are affected. The method `refreshScene()` is used to re-render

---

9 Although not all implementations of such systems would separate the graphical user interface code from the code handling the core functionalities in this way, a strong argument can be made to develop components in this way to facilitate monitoring [59].

10 Java iterators require that `Iterator.remove()` be used to remove an item from the collection; removing directly from the collection can cause undefined behavior.
final protected void eraseNode(int nd) {
    Node n = (Node)(nodes.get(nd));
    n.unDraw(sc); nodes.remove(nd);
    for (Iterator iter = edges.iterator(); iter.hasNext(); ) {
        Edge e = (Edge)(iter.next());
        if (e.from()==n || e.to()==n) {
            e.unDraw(sc);
            iter.remove(); // Removes e from edges
        }
    }
}

final protected void eraseEdge(int ed) {
    Edge e = (Edge)(edges.get(ed));
    e.unDraw(sc); edges.remove(ed);
}

final protected void refreshScene() {
    sc.clear(); // Clears the Scene
    for (Iterator iter = nodes.iterator(); iter.hasNext(); ) {
        Node n = (Node)(iter.next()); n.draw(sc); }
    for (Iterator iter = edges.iterator(); iter.hasNext(); ) {
        Edge e = (Edge)(iter.next()); e.draw(sc); }
}

Figure 3.6: DiagEditor class code (part 2)

all of the nodes and edges of the diagram. It does so by first clearing the Scene, and then iterating through the nodes and edges vectors, drawing the elements one by one.

In the final section of code for the DiagEditor class, shown in Fig. 3.7, we first have the methods registerNode() and registerEdge(). These methods are used to register the specialized Node and Edge class prototype objects when the DiagEditor is being initialized. The process of registration includes adding the object to the n_types or e_types vector, and adding its button to the graphics editor window. The next
final protected registerNode(Node n) {
    n_types.add(n);
    . . . Create n.className() button on diagram editor for n . . .
}
final protected registerEdge(Edge e) {
    e_types.add(e);
    . . . Create e.className() button on diagram editor for e . . .
}
final public void setUp() {
    . . . Create application window . . .
    registerAll();
    . . . Finish initializing app window . . .
}
protected void registerAll() {
    // This method should be redefined in a derived class
    registerNode(new Node()); registerEdge(new Edge());
}

Figure 3.7: DiagEditor class code (part 3)

method, setUp(), is used to initialize the DiagEditor by creating the application window, registering the nodes and edges (by calling registerAll()), and displaying the window.

The last method shown, registerAll(), is used to register all of the specialized Node and Edge types by simply calling registerNode() and registerEdge(), and is intended to be overridden in a derived class of DiagEditor. Sample code for this method is given in Fig. 3.7 just to illustrate what the method definition should look like in the context of the base class DiagEditor. In a specialized application, for example the one shown in Fig. 3.1 that uses Circle and Square nodes and Solid_Line and Dotted_Line edges, the code for registerAll() would look like this:

    registerNode(new Circle()); registerNode(new Square());
It should be emphasized that all of the code in the DiagEditor class, with the exception of registerAll(), remains the same for any application built upon the framework. Adding, moving, and erasing things from the diagram (as well as handling mouse clicks) is done in the same way from application to application; only the hook methods found in the specialized Node and Edge classes will vary, along with the objects that must be registered with the DiagEditor when it is initialized.

3.2.3 Specifying the Diagram Editor

Let us now consider how we may specify the behavior of the diagram editor framework—specifically the handler methods of the DiagEditor class we presented in the previous section. We will give specifications that outline the important behaviors of these methods, but not go to the extreme of using “complete” specifications as in Sec. 2.4 since our primary focus is monitoring, and not formal correctness proofs.

In Fig. 3.8, we start with the addNode() and addEdge() methods. For addNode(), the precondition stipulates that nt is a valid index of a prototype node in the n_types vector, and the point p is in the drawing area defined by the Scene sc. Because we are not concerned with the details of the Scene, we do not give a definition for the DrawingArea() function in the specification.

The next part of the specification with the pres prefix, pres.addNode(nt, p), gives us a list of the variables preserved by the method. A class data member or method parameter is preserved if its value at the time of the method invocation is the same as the value when the method completes execution. The notion of “sameness” that we
Figure 3.8: Specifications for the DiagEditor add methods

use in preserves clauses is one-deep equality for objects, and value equality for simply-typed variables. We define one-deep equality as follows. Two container objects are one-deep equal if the references they contain are equivalent (i.e., equal with respect to the ‘==’ operator), and are in the same order if the container objects have a natural ordering. Non-container objects are one-deep equal if the references to their corresponding non-private data members are equivalent. During monitoring, the comparison is made between the current value of the object, and that of a one-deep copy of the object that was created immediately before the method was executed.

Here, the vectors edges, n_types, and e_types are preserved by addNode(). This means that the list of references they contain remain unchanged by the method. This does not mean, however, that the state of the individual elements of these vectors will
necessarily remain unchanged. We also state that the parameter \( p \) will be preserved, since its individual \( x \) and \( y \)-coordinates will not be changed by the method. However, we do not state that the parameter \( nt \) is preserved. This is because it is a simple integer, and will automatically be preserved from the standpoint of the caller since Java uses pass by value for simply-typed variable parameters.

An alternative semantics for preserves clauses that may seem reasonable is to use the provided \texttt{equals()} method for the given class to check for equality. Although this would also give us a notion of sameness based on value-equality instead of reference-equality, it would cause the specification semantics to be dependent on the implementation of \texttt{equals()}. With such a definition, preservation would often hinge on whether the user has provided an implementation of \texttt{equals()} for the given class, or if the \texttt{equals()} method is simply inherited from a parent class. Furthermore, if we compare an object to a one-deep copy of its previous state using \texttt{equals()}, we may not get equality if \texttt{equals()} checks for “deep-equality” that goes further down than one level of references. To ensure a consistent and easy-to-understand semantics, we have chosen the one-deep definition of equality for preserves.

In the postcondition for \texttt{addNode()}, we start by giving a definition for the specification variable \( n \). Such definitions can make for much easier reading and stating of specifications, as well as evaluating during monitoring. Here, \( n \) is defined as the (reference to the object) result of the first hook method call, which happens to be \texttt{clone()} on the \( nt+1 \)st item of \texttt{n_types}, the prototype object for the given \texttt{Node} type to be added to the diagram. The line after the definition first says that the reference value \( n \) will be appended to the \texttt{nodes} vector of references. Again, this is not saying that the \textit{state} of the corresponding nodes are necessarily the same. As in the preserves clauses,
we use one-deep equality semantics when speaking of value equality. The remainder of this line states that n’s center will set to p (the use of ‘==’ means reference equality), and that two hook method calls will be made by `addNode()`. The next two lines refer to the first hook call, recorded in $\tau[0]$, which is the call to `clone()` we mentioned. In the second line, we stipulate that the class of n be the same as that of the cloned prototype `Node` object using the `Class()` function\(^{11}\). The final two lines of the postcondition refer to the second hook call, this one made to `draw()` on the new `Node` n. Note that in addition to the object reference of the hook target ($\tau[1].target$) being given, the object state of the target before the call ($\tau[1].pre.target$) is also specified. The input object state is specified as being the same as the object state for n at the end of `addNode()`’s execution, which ensures that its center is at p before `draw()` is called. Also note that we say nothing about how `sc` is modified by `addNode()`, just that it is the argument to the hook call to `draw()`. Although the details of `Scene` and the implementations of the `draw()` and `unDraw()` methods in the `Node` and `Edge` classes are not given, with proper specifications for them, we could “plug” them into this trace-based postcondition to arrive at how `sc` would be changed by `addNode()`.

Before moving on, we should note something that is not stated here in the postcondition. Although we have said things such as the references inside of `edges` stays the same, we have not said that the states of the objects inside of `edges` remain unchanged. This condition, although it should be true for a proper implementation of a diagram editor, is not something that can be checked by the TCGen monitoring.

\(^{11}\)Because it is stated here in the postcondition, the TCGen monitor will necessarily check this condition when `addNode()` returns. However, if the postcondition for `Node.clone()` states this same condition, and those specifications are input into TCGen, the monitor would also check it immediately after the call to `clone()` returns. To prevent such redundant checking, this condition could be removed from one of the class specifications.
tool. This is because TCGen makes a one-deep copy as mentioned before of the fields of the class object this for saving state. While this scheme will give us a vector containing the same references in edges, it does not give us a vector with copies of the state of each of the individual elements of edges. Further details about the monitors generated by TCGen will be discussed in Sec. 3.3.1.

The specifications for addEdge() are very similar, except that instead of fixing the new Edge e at a particular point, we specify what Nodes among nodes that e is attached to. Again, the method addEdge() is specified to make two hook calls, the first to clone() on the given prototype Edge object, and the second to draw() to render the edge.

In Fig. 3.9, we show the specification for moveNode(). The precondition states that nd should be a proper index for the nodes, and that p should be in the drawing area defined by the Scene. The preserves listing tells us that nothing should be added or removed from the collections nodes, edges, n_types, and e_types. The postcondition starts with three definitions, the first defining n to be the node that will be moved, the second adj_e being the sub-vector of edges containing exactly those Edge’s adjacent to n, and the third ln simply being the length of adj_e. The next line of the postcondition tells us that the center of n at the end of execution will be the Point p, and also that there will be 2+2+ln hook method calls. The next two lines dealing with τ[0] tell us that the first hook method call will be to undraw() on the object n with argument sc. It is also true that the state of n at the time of the hook call will be same as its state at the beginning of the call to moveNode(), but because the pre-state of n is not saved by the monitoring system (only the reference to n is saved as part of the one-deep copy of nodes), we do not specify this condition. The next three lines dealing with
\[ \text{pre.moveNode}(nd, p) \equiv 0 \leq nd < \lvert \text{nodes} \rvert \land p \in \text{DrawingArea}(sc) \]
\[ \text{pres.moveNode}(nd, p) \equiv \text{nodes}, \text{edges}, \text{n_types}, \text{e_types} \]
\[ \text{post.moveNode}(nd, p) \equiv \text{Node } n \overset{\text{def}}{=} \text{nodes}_{nd} : \]
\[
\begin{aligned}
\text{Vector } \text{adj.e} & \overset{\text{def}}{=} \langle \text{Edge } e \in \text{edges} : e.\text{from} == n \lor e.\text{to} == n \rangle : \\
\text{int } l & \overset{\text{def}}{=} \lvert \text{adj.e} \rvert : \\
n.\text{center} & == p \land \lvert \tau \rvert == 2 + 2 \times l \\
\land \tau[0].\text{method} & == \text{"unDraw(Scene)"} \\
\land \tau[0].\text{target} & == n \land \tau[0].\text{arg}_1 == \text{sc} \\
\land \tau[1 + l].\text{method} & == \text{"draw(Scene)"} \\
\land \tau[1 + l].\text{target} & == n \\
\land \tau[1 + l].\text{pre.target} & == n \land \tau[1 + l].\text{arg}_1 == \text{sc} \\
\land ( \forall i : 0 \leq i < l : \tau[1 + i].\text{method} == \text{"unDraw(Scene)"} \\
\land \tau[1 + i].\text{target} & == \text{adj.e}_i \land \tau[1 + i].\text{arg}_1 == \text{sc} \\
\land \tau[2 + l + i].\text{method} & == \text{"draw(Scene)"} \\
\land \tau[2 + l + i].\text{target} & == \text{adj.e}_i \\
\land \tau[2 + l + i].\text{arg}_1 & == \text{sc} \\
\land \tau[2 + l + i].\text{pre.target} & == \text{adj.e}_i ) \\
\end{aligned}
\]

Figure 3.9: Specification for \text{moveNode()}

\[ \tau[1 + l] \] tell us that after the hook method calls to \text{unDraw()} are made on the node and its adjacent edges, the node \( n \) will be redrawn with the hook call to \text{draw()}. Note that here, the pre-state of the target \( n \) before the hook call is the same as its ending state; \( n \) has had its center moved to \( p \) before it is re-rendered. Because this state is accessible at the end of \text{moveNode()}’s execution, we can say that the pre-state of the hook call’s target object is the same as \( n \)’s state.

Now, the hook method calls made to erase and re-render the adjacent edges are taken care of in the next six lines with the quantified statement. In the first two of these lines, the hook calls numbered 1 through \( l \) on the trace \( \tau \) are made to \text{unDraw()} on each of the elements of the vector \text{adj.e}. As before, the hook call is made on the
pre.eraseNode(nd) ≡ 0 ≤ nd < |nodes|

pres.eraseNode(nd) ≡ n_types, e_types

post.eraseNode(nd) ≡ Node n = def #nodes

Vector adj.e = def (Edge e ∈ #edges : e.from == n ∨ e.to == n):

int ln = def |adj.e|:

nodes = (Node d ∈ #nodes : ¬d == n)
∧ edges = (Edge e ∈ #edges : ¬e.from == n ∧ ¬e.to == n)
∧ |τ| = 1 + ln
∧ τ[0].method = “unDraw(Scene)”
 ∧ τ[0].target == n ∧ τ[0].arg1 == sc
∧ (∀ i : 0 ≤ i < ln : τ[1 + i].method = “unDraw(Scene)”
  ∧ τ[1 + i].target == adj.e_i ∧ τ[1 + i].arg1 == sc)

pre.eraseEdge(ed) ≡ 0 ≤ ed < |edges|

pres.eraseEdge(ed) ≡ nodes, n_types, e_types

post.eraseEdge(ed) ≡ Edge e = def #edges

edges = (Edge d ∈ #edges : ¬d == e) ∧ |τ| = 1
∧ τ[0].method = “unDraw(Scene)”
 ∧ τ[0].target == e ∧ τ[0].arg1 == sc

Figure 3.10: Specifications for the DiagEditor erase methods

given adjacent edge, but we do not state the fact that the target’s pre-state is the edge’s pre-state at the beginning of moveNode()’s execution, because this information will not be stored by the monitoring system. The last four lines are similar, going through hook calls ln+1 through 2+2*ln, where the adjacent edges are re-rendered via the draw() method. The pre-states of these edges at the beginning of these hook calls are the same as their states at the end of moveNode()’s execution, and so we can state that the target’s pre-state is the same as adj.i’s ending state.

The specifications for eraseNode() and eraseEdge() are shown in Fig. 3.10. Starting with eraseNode(), the precondition states that nd should be a valid index in nodes, and the preserves list says that n_types and e_types will remain unchanged. The
postcondition of \texttt{eraseNode() has many similarities to that of moveNode() so we will}
not go into as much detail here. First, definitions are provided for \( n \), \( \text{adj.e} \), and \( \text{ln} \)
as before, except it is important to note that these definitions are in relation to the
\textit{beginning} values of \texttt{nodes} and \texttt{edges}, not the \textit{ending} values. This is because the final
values of these vectors will not contain the deleted node and edges. The next two
lines simply state that \( n \) and the edges of \( \text{adj.e} \) will be removed from \texttt{nodes} and \texttt{edges},
respectively. The next line says that \( 1+\text{ln} \) hook calls will be made. The next two
lines after that says that the first hook call will be to \texttt{unDraw()} on the node \( n \), and
the last two lines say that \texttt{unDraw()} will be invoked on each of the adjacent edges of
\( n \). Turning to the specification for \texttt{eraseEdge()}, we see that it is much simpler since
erasing an edge will not require adjacent nodes to also be erased.

### 3.3 Monitoring Trace-Based Specifications

Now that we have presented trace-based specifications for the handler methods of
the \texttt{DiagEditor} class, we will turn to our main question:

\textit{How can we check if a trace-based specification is violated?}

Checking for violations of standard black-box specifications during monitoring and
testing is not much of an issue. In such a scheme, the tester would be able to pro-
vide inputs, run the methods to be checked, and then evaluate the observed outputs
against the postcondition specifications. However, trace-based specifications contain
information that goes beyond the pre- and post-states of the objects being used: they provide information about how the method interacts with other methods via
the trace variable \( \tau \). This information is critical not only from a formal reasoning
standpoint, as was seen in Chapter 2, but also in capturing the \textit{essential behaviors}
of the methods themselves outside of a specific context. In other words, template methods are primarily characterized by the method calls they make, and not by the state changes they cause [33, 24]. This is especially true for frameworks, which make heavy use of template methods. We have presented a way to precisely specify such characterizations, but we are still left with the problem of monitoring these vital behaviors.

The key problem we face in monitoring trace-based specifications is that we cannot wait until the template method finishes execution to try to record information about the calls made to the hook methods. What we need to do instead is to intercept these hook method calls during the template method’s execution as they are made. As mentioned in Sec. 3.1, one possibility would be to introduce a trace variable to the system, and modify the code to be monitored in such a way that the trace would be updated as needed. However, problems will arise if such instrumentation is not done carefully and consistently, since the system itself is being modified. This can be a difficult task indeed if done manually. Instrumentation is not even an option, though, if the source code is not available. This is often the case when proprietary software is being used, and frameworks often fall within this category.

The answer comes from the same mechanism that allows us to enrich the behavior of template methods such as the handler methods of the diagram editor through the redefinition of hook methods, i.e., polymorphism. Rather than intercepting these calls by modifying already existing source code (which we may not have access to in the first place), we can redefine the hook methods so that they update the trace appropriately whenever they are invoked. We can do this by creating special “trace-saving” classes that accomplish this task. We have developed a tool, TCGen, that automates the
process of creating these classes, along with support classes to help achieve this aim. We will first go through the details of how the generated trace-saving and support classes work in Sec. 3.3.1, walk through a sample run in Sec. 3.3.2, and then look at how to run the tool in Sec. 3.3.3.

3.3.1 Monitoring with Trace-Saving Classes

Before going into the details about how polymorphism can be leveraged to allow for capturing method calls, we first describe some of the data structures that we use for storing trace information. Not surprisingly, we use a Trace class to store sequences of method calls. Trace objects contain a Vector of trace records of type TraceRec, where each record holds information about a single hook call. However, different hook methods may have different numbers and types of parameters, and some may return values when others do not. This means that the information that needs to be stored for each call may vary depending on which hook method is called. To accommodate these differences, TCGen generates a special trace record class, which is a derived class of TraceRec, for each hook method of each class. These specialized trace records are equipped to store references, as well as state, of the target object and the method parameters.

Another issue concerns the possibility of template methods also being hook methods, and the need to track more than one trace during execution. Recall that for a given call to a template method t(), the trace variable associated with its execution should only record those hook methods that t() invokes, either directly, or via other non-hook methods. However, if t() calls another method t2() which is both a hook
and a template method, t()'s trace should not contain those hook calls made by t2(). These hook calls made by t2() should instead be recorded on a trace associated with t2()'s execution, and thus, a second trace would be needed. Because of the possibility of chains of such calls, where methods that are both hooks and templates are calling other such methods, we have to consider storing multiple trace variables simultaneously during monitoring. This situation is handled by using a stack of traces, TraceStack, where at any given moment during execution, the topmost Trace object on the stack is the trace associated with the method whose calls are currently being tracked—i.e., the method that is actively running, or has initiated a current chain of calls to non-hook methods. Only a single TraceStack is used by the entire system during monitoring, and so this class is defined as a singleton class with all static methods.

These classes, along with the specialized trace record classes for some of the hook methods of the Node class (specifically draw, isOn(), and clone()), are shown in the UML diagram in Fig. 3.11. Starting with the TraceStack class, we see that its methods and its stack variable are all static (which is denoted in UML via underlining). We have the usual push() and pop() methods for stacks, as well as an appendToTopTrace() method which allows us to add records to the current trace. The Trace class, in addition to its calls vector, also keeps track of the number of call records it has stored in n_calls, as well as a special record caller of the initiating call if the initiator was a hook. The initiating call is simply the method call for whom the trace was created. This caller variable is used primarily to store the pre-state for trace-saving purposes, as we will see shortly. The base TraceRec class stores only the target object reference.

However, because t2() is still a hook, the fact that t() did invoke t2() should still be recorded on t()'s trace.
and the name/signature of the method, whereas the specialized derived classes store the references to the parameters, pre-state of the target and parameters, and the reference and state of the result, if the method returns a value. In these specialized classes, the field names mimic those of the variables found in the specifications. Note that variables that store parameter post-state were not included in these classes. This design decision was made because these values generally can be inferred from the pre-state (barring nondeterminism), and thus serve a limited use in specification-based monitoring. Furthermore, the postconditions of the hook methods recorded on the trace will themselves be checked during monitoring, and so this post-state will undergo a sort of correctness check regardless (though not with respect to the trace).

Now we turn to the discussion of how the necessary information is recorded in these classes. To help accomplish this, TCGen generates special trace-saving classes for each of the framework classes that contain methods flagged as hook methods.
in the specifications. For monitoring the **DiagEditor** framework, we track the calls to the hook methods in the **Node** and **Edge** classes using the trace-saving classes **TS_Node** and **TS_Edge** (the “TS” prefix stands for “trace-saving”). We show the code for **TS_Node** in Fig. 3.12; the code for **TS_Edge** is virtually the same, and is thus omitted. During monitoring, we use specialized **TS_Node** and **TS_Edge** objects for our **Nodes** and **Edges** in the **DiagEditor**, so that whenever hook methods such as `draw()` or `unDraw()` are invoked on one of the diagram elements, the call is dispatched to those found in these trace-saving classes. To do this, when we create an instance of **DiagEditor** for monitoring, we register a **TS_Node** object and a **TS_Edge** object as the types of **Nodes** and **Edges** that the diagram editor should use. Here, we are using the same registration mechanism that allows us to register a **DiodeNode**, a **TransistorNode**, a **SimpleWireEdge**, etc. when creating a circuit editor application\(^\text{13}\). Essentially, the **DiagEditor** that we instantiate for monitoring the framework is itself an application—the specialized **Node** and **Edge** classes in this case are tailored for saving trace information. This is illustrated in the UML diagram in Fig. 3.13.

For this to work, **TS_Node** must be a derived class of **Node** (and **TS_Edge** a derived class of **Edge**). In **TS_Node**, the hook methods should be redefined so that they update the trace stack and current trace, while still invoking the original methods. In these redefined hooks, the method’s precondition (if given) is first checked; this helps in locating bugs, where the problem may be with the caller of the hook rather than the hook method itself. Then, the state of and references to the fields, as well as the parameters’ state, are saved using `auxClone()` methods which we will describe later.

\(^\text{13}\)Here, we are not talking about how to monitor applications of the framework, but just the base framework itself. If we wanted to monitor such applications, we would instead create trace-saving classes such as **TS_DiodeNode** that would extend **DiodeNode**, and register those with the framework.
class TS.Node extends Node {
    protected void draw(Scene sc) {
        System.out.println("Method Node.draw called.");
        if (/* draw's precondition violated */)
            System.out.println("Precondition of draw not met!");
        Node.pre_this = Util.auxClone(this); // saves refs. to data members
        Point.pre_center = Util.auxClone(center);
        Scene.pre_sc = Util.auxClone(sc);
        Node.draw_TraceRec caller = new Node.draw_TraceRec(this, sc);
        Trace tau = new Trace(caller); // create new trace w/ draw as caller
        TraceStack.push(tau);
        super.draw(sc); // call original hook method
        if (/* draw's preserves clause violated */) 
            System.out.println("Preserves clause of draw violated!");
        if (/* draw's postcondition violated */) {
            System.out.println("Postcondition of draw not met!");
            tau.print();
        }
        TraceStack.pop();
        caller.addPostVals(this, sc);
        TraceStack.appendToTopTrace(caller);
    }

    unDraw(), className(), isOn(), and boundPt() are all redefined in the same way as above; 
    clone() is handled a little differently...
}

Figure 3.12: TS.Node class with redefined hook draw()
Next, a new specialized `TraceRec` object `caller` is created to represent this hook call, and it is used to initiate a new trace `tau`, which itself is pushed onto the stack. It is important to differentiate between this `tau`, which is used to record hook methods that this hook (`draw()` in this case) makes, and the trace that will be directly under it on the `TraceStack`, which will record this particular hook call. Then, we call the original hook definition, which is done using the `super` mechanism of Java. When the original hook is finished executing, we will have the old (saved) field and parameter state values in hand, along with their current values. Furthermore, we will have the updated trace `tau`, which represents the trace variable $\tau$ mentioned in the specifications. We use these values to evaluate the preserves clause and postcondition of the hook method (if they are given). After that, we can discard the topmost trace by `popping` it off of the `TraceStack`, and then update the current trace now at the top by appending an updated `caller`.

All of the hook methods are handled in the same way, with one exception—the `clone()` method. The problem with `clone()` is that the definition given in the base
protected Object clone() {
    ... check precondition, save pre-state, initialize tau, and update TraceStack as before ...
    TS_Node result = this.cloneProxy();
    ... check preserves clause and postcondition, update caller and TraceStack as before ...
    return result;
}

public TS_Node cloneProxy() { // User may have to redefine this
    Node base_result = this.super.clone();
    TS_Node result = new TS_Node();
    result.center = base_result.center;
    return result;
}

Figure 3.14: Cloning methods in TS_Node class

class will return a Node object, and not a TS_Node object, and so the returned object cannot be tracked by the monitoring system. Here, this would mean that none of the Nodes added to the diagram would be of type TS_Node, since they are all produced by cloning the registered prototype Node. Thus, none of the subsequent calls to hooks such as draw() would ever be intercepted during monitoring. What is needed is a version of clone() for TS_Node objects which specifically produces TS_Node objects, and not just Node objects. TCGen does this, as shown in Fig. 3.14. TCGen generates a cloneProxy() method which is called by the redefined clone() hook method, instead of the super.clone() method defined in its parent class. The cloneProxy() method itself calls super.clone(), but instead of returning the returned base_result object—which is a Node but not a TS_Node—it creates a new TS_Node result, and copies all of the field references from base_result to result, and returns the result object. The advantage to
class Node_draw_TraceRec extends TraceRec {
    public Node _pre_target;
    public Scene _arg1, _pre_arg1;

    public Node_draw_TraceRec(Node _this, Scene sc) {
        method = "Node.draw(Node,Scene)"; // method name/signature
        target = _this; // reference copy
        _pre_target = Util.auxClone(_this); // state copy
        _arg1 = sc; // reference copy
        _pre_arg1 = Util.auxClone(sc); // state copy
    }

    public void addPostVals(Node _this, Scene sc) {
        // Current TCGen impl. only saves returned result info, nothing to do here
    }
}
using \texttt{auxClone()} methods\textsuperscript{14} which we will describe shortly. Recall that in the hook methods defined in the trace-saving classes, after the original hook method returns, \texttt{updatePostVals()} is invoked to add the information about the result of the call. Again, the current implementation of TCGen does not save the post-state of the target and parameters, so the only information that would be saved by \texttt{updatePostVals()} would be the reference and state of the returned object, if the hook does return an object at all. However, note that the code in Figs. 3.12 and 3.15 still passes the target object and parameters as arguments. This is done to facilitate the saving of their post-state in the trace record in cases where the user does want to store such information; the only changes that would be needed would be to manually add the extra data members to the specialized \texttt{TraceRec} class, and add the update code in \texttt{updatePostVals()}.

Now, let us discuss the \texttt{auxClone()} methods. TCGen generates a special \texttt{Util} class which contains these methods which allow us to make one-deep copies of objects in the framework. For each class encountered during parsing of the specifications, an \texttt{auxClone()} method body is constructed in the following manner. First, a new object \texttt{result} of the same type of the argument \texttt{obj} to \texttt{auxClone()} is created. Next, if \texttt{obj} is of a container type, the references in \texttt{obj} are simply copied over into \texttt{result} (and in the same order if the elements of \texttt{obj} have a natural ordering). If \texttt{obj} is not of a container type, then each of its non-private field values (the field information is obtained using Java reflection) are copied into the corresponding fields in \texttt{result} using simple assignment. In either case, \texttt{auxClone()} returns the \texttt{result} object after the copying is complete.

\textsuperscript{14}If we used \texttt{clone()} methods provided by the system instead of \texttt{auxClone()} methods here, it may cause problems during monitoring, because those calls to \texttt{clone()} may be unintentionally intercepted by the monitoring system, and they may also not provide the same copy semantics as those intended in the specifications.
The Util class also serves another purpose: to provide auxEquals() methods which check if two objects are equal using one-deep equality semantics. These methods are generated by TCGen in a similar fashion as their corresponding auxClone() methods, where container objects are equal if they contain the same references, and non-container objects are equal if the references in their non-private fields are equivalent.

Now let us look at the main testing class, `Test_DiagEditor`, in Fig. 3.16. Here, the class `Test_DiagEditor` acts as an application built on the `DiagEditor` framework, with additional testing methods added. First, we need to register a `TS_Node` and a `TS_Edge` object in the redefined registerAll() method—because the registration mechanism can vary from framework to framework, TCGen requires the user to manually fill in the code for such methods. For each of the methods of `DiagEditor` whose specifications are given, TCGen will generate a test method that will take the this object as the test case object, as well as parameters mirroring those found in the parameter list of the original method. The checking code generated in the test methods is similar to the checking code found in the redefined hooks in the trace-saving classes (Fig. 3.12). Let us use the `test_eraseNode()` method as an example, while looking at `eraseNode()`’s specifications in Fig. 3.10. Each test method will first make sure the inputs constitute a valid test case, i.e., they do not violate the method’s precondition. The method `eraseNode()`’s precondition states that `nd` is a valid index for `nodes`. We should reiterate that the variables’ data members that are referred to in the specifications should be either public or protected; if they are private, then they should not be visible to clients or those who wish to extend the class, and therefore should not be visible in the specifications. By having the testing class be a derived class of the framework, and
class Test_DiagEditor extends DiagEditor {
    protected void registerAll() {
        registerNode(new TS_Node()); registerEdge(new TS_Edge());
    }

    public void test_eraseNode(int nd) {
        if (0 <= nd && nd < nodes.length()) {
            DiagEditor _pre_this = Util.auxClone(this); // saves refs. to data members
            Vector _pre_nodes = Util.auxClone(nodes);
            Vector _pre_edges = Util.auxClone(edges);
            Vector _pre_n_types = Util.auxClone(n_types);
            Vector _pre_e_types = Util.auxClone(e_types);
            Scene _pre_sc = auxClone(sc);
            Trace tau = new Trace(); // no caller/initiator is stored for test meths.
            TraceStack.clear(); TraceStack.push(tau);
            eraseNode(nd); // call method to be tested
            if (!Util.auxEquals(n_types, _pre_n_types) || !Util.auxEquals(e_types, _pre_e_types)) {
                System.out.println("Preserves clause of eraseNode violated!");
            } else {
                boolean post_result;
                ... postcondition eval. code, see Fig. 3.17 ... 
                if (!post_result) {
                    System.out.println("Postcondition of eraseNode not met!");
                    tau.print(); }
            }
        } else {
            System.out.println("Precondition of eraseNode not met!");
        }
    }

    ... test methods for other methods of DiagEditor ...
}

Figure 3.16: The testing class Test_DiagEditor

including it in the same package as the framework classes, the necessary data members
are visible to the testing class.

After the precondition is checked, copies of the state of this, the data members,
and the method parameters are stored in special _pre objects using the generated
auxClone() methods. Note that by creating a one-deep copy of this in this fashion, we get the initial references to the data members of the current class object. Also note that since the parameter nd is a simply typed int, there is no need to save its old value which necessarily will be the same at the end of eraseNode()’s execution. Next, we initialize a new trace object tau, clear the TraceStack, and push tau on it.

At this point, we can then execute the method to be checked with the given parameters. Because the Nodes and Edges will all be of type TS.Node or TS.Edge, all calls made to hook methods will be dispatched to the trace-saving versions defined in the derived classes. Here, when execution of eraseNode() finishes, all of the trace information will be stored in tau. We then check to see if all of the variables listed in the preserves clause are “one-deep equal” to their old values using the auxEquals() methods. Next, we evaluate the postcondition, which has been translated into executable code by TCGen. The actual code for evaluating eraseNode()’s postcondition is shown in Fig. 3.17. If the postcondition is violated, a message is printed to the screen, along with the value of the current trace.

Now, let us look at the postcondition evaluation code in Fig. 3.17. First, the definitions given at the start of the postcondition are implemented as code. For eraseNode(), the definitions for n and ln are relatively straightforward to translate, whereas adj.e is defined as a sub-vector of .pre_edges whose elements have to meet a certain condition. For such definitions, TCGen generates a for-loop that iterates through the items of the base container object (in this case, .pre_edges), and then adds them to the new object if they meet the given condition.
/* Definition evaluation code */
Node n = _pre_nodes.get(nd);
Vector adj_e = new Vector();
for (Iterator _iter1 = _pre_edges.iterator(); _iter1.hasNext()) {
    Edge e = (Edge)(_iter1.next());
    if (e.from == n || e.to == n)
        adj_e.add(e);
}
int ln = adj_e.length();

/* Preliminary evaluation code for top-level expression */
Vector _vec1 = . . ._pre_nodes not equiv. to n; defn similar to adj_e above...
Vector _vec2 = . . ._pre_edges not adj. to n; defn similar to adj_e above...
boolean _bool1 = true; // Defn. for universal quantifier expr. in last 2 lines
for (int i = 0; i < ln && _bool1; i++) {
    _bool1 = tau.get(1+i).method.equals("unDraw(Scene)")
    && tau.get(1+i).target == adj_e.get(i)
    && ((Edge_unDraw_TraceRec)tau.get(1+i)).arg1 == sc;
}

/* Top-level expression evaluation */
post_result = Util.auxEquals(nodes, _vec1) && Util.auxEquals(edges, _vec2)
    && tau.length() == 1 + ln && tau.get(0).method.equals("unDraw(Scene)")
    && tau.get(0).target == n
    && ((Node_unDraw_TraceRec)tau.get(0)).arg1 == sc;

Figure 3.17: Evaluating the postcondition of eraseNode()

The next part of the postcondition checking code is the preliminary evaluation block for the entire postcondition. Because the boolean expressions in our specifications can contain quantified expressions, as well as special definitions for defining subsets and sub-vectors, the code to evaluate these boolean expressions may take up several lines of statements. A simple boolean expression—one that does not contain quantifiers or special definitions—can be translated into a single Java boolean expression. However, non-simple expressions have their (non-nested) quantified expressions
and special definitions evaluated in a preliminary evaluation block, and their values are assigned to substitution variables. These variables are substituted into the translated boolean expression, which is evaluated after the preliminary block.

Although nested quantifications and special definitions are not found in our example, TCGen does allow for such nesting in specifications. The key to how this is done hinges on the fact that quantifications and special definitions are evaluated using for-loops, where the indexing is handled in the initial for( ; ; ) part, and the boolean condition part is evaluated in a block of code inside of the loop body. To illustrate how nesting is handled, suppose $e$ is a non-simple expression that is nested inside of a quantification or special definition $d$, and $C_e$ is the evaluation code for $e$ which assigns $e$’s value to a substitution variable $v_e$. Now, suppose $B_d$ is the body code of the for-loop that would be used to evaluate the condition part of $d$ if $e$ was a simple expression, and $v_e$ is substituted into $B_d$ where the value of $e$ is needed. Thus, to evaluate $d$, we just need to use $C_e; B_d$ as the body for $d$’s for-loop.

Returning to our example, we see that there are two special definitions in the specifications, one that gives the sub-vector of #nodes that are not (reference) equal to $n$, and another that gives the sub-vector of #edges that are not adjacent to $n$. The code for these is very similar to the code for adj.e, and is thus omitted. There is also a universally quantified expression in our postcondition, whose evaluation code is shown in the preliminary block. To handle universal quantification, TCGen first sets the substitution variable equal to true, and then iterates through the indices using a for-loop, checking the condition each time, and exiting the loop whenever the condition is not met. The final statement of the postcondition code is the evaluation
of the postcondition expression itself, with the proper substitutions made, and the assignment of its value to the variable \texttt{post\_result}.

We should note that in the conditions involving \texttt{arg1} in the translated specifications, that the accessed \texttt{TraceRecs} on \texttt{tau} need to be casted to their proper specialized \texttt{TraceRec} classes; this is because the base \texttt{TraceRec} class only has the data members \texttt{target} and \texttt{method}, but not \texttt{arg1}. This casting must either be made explicit in the specifications (which is not shown in Figs. 3.8–3.10), or added manually in the monitoring code. In previous implementations of TCGen, a \texttt{TraceRec} class that included a \texttt{Vector} that held onto the \texttt{arg} references was used for recording all of the hook calls—no specialized \texttt{TraceRec} classes were used, and so such casting was unnecessary. The use of customized \texttt{TraceRecs} began when we chose to allow for saving parameter state in later implementations.

In addition to pre- and postconditions, and a list of preserved variables, the user can also include a \textit{signals} clause [54] in a method’s specification that can be checked by TCGen. A signals clause is similar to a postcondition, but specifies what conditions must hold when a specific type of exception has been thrown by the method, instead of when the method returns normally. The main changes to the code, as shown in the examples in Figs. 3.12 and 3.16, is that the call to the method being checked (along with the preserves and postcondition checking code afterwards) is enclosed in a \texttt{try}-block, and a catch block for each specified exception type is provided that checks the condition that should be true when its associated exception has been thrown. In the redefined hooks, each of the catch blocks also provides code that updates the trace, and re-throws the caught exception to allow for the framework to proceed as it would if it were not being monitored.
Previously, we mentioned that the methodology that we presented could also be applied when the base classes that are to be specialized by application developers are abstract. We do so with only slight modifications to the above process. First, note that the abstract methods of such a class can be redefined (and indeed, must be), and are therefore hooks. When creating the trace-saving version of the class, however, there are no concrete definitions for these abstract methods in the base class, and so no super-call should be made. In the place of the super-call, a call to a minimal implementation of an auxiliary version of the hook method should be made. Here, we mean “minimal” in the sense that the method definition, which needs to be hand-coded by the user, adheres to its base class specification and allows the framework to function properly. In situations where we are only concerned with checking specifications of the call sequences, and not the state changes of the objects in the framework, a call to an auxiliary method in a redefined hook can be omitted if it is only responsible for changing object state\textsuperscript{15}. In a sense, these trace-saving classes are acting as stub classes—concrete classes extending abstract classes that are acting as stand-ins only for testing/monitoring purposes.

In this section, we have described a methodology for monitoring frameworks using trace-saving classes. However, we have not mentioned how we might monitor applications that may be built on such frameworks—specifically, how we could capture trace information during the execution of applications. Here, we can again employ the same techniques that we used in monitoring frameworks, given that the framework’s hook methods to be tracked have not been made final in the application code,

\textsuperscript{15}This is not the case for a hook that is also a template method, because its calling behavior is also important. In this case, an auxiliary version of the hook that makes these necessary calls should be implemented and used by the redefined hook.
and that we can still register trace-saving objects. Essentially, the application to be monitored must still be customizable, and can still be treated as a framework in that sense. For the circuit editor application, for example, we would create trace-saving classes such as TS_DiodeNode for tracking calls to the hooks in the DiodeNode class, and register those objects with the editor by appropriately defining the registerAll() method in the testing class.

3.3.2 Walking through a Monitoring Run

Let us see how Test_DiagEditor.test_eraseNode() works using an extended sequence diagram [14] in Fig. 3.18. For our sample execution run, we will assume a diagram where the node with index nd in the nodes vector is adjacent to exactly one edge, the Edge at index j in the edges vector. In the figure, the first two vertical lines grouped under the heading this together represent the test case object. The two individual lines under the this heading, labeled Test_DiagEditor and DiagEditor, represent the classes whose code is used to define objects of type Test_DiagEditor. Recall that Test_DiagEditor is a derived class of DiagEditor, and inherits all of its methods, including eraseNode(). We use separate lines here to help show how the code of these two classes interact during execution. The next two vertical lines grouped under the heading nodes represent the nd+1st node in the nodes vector—the Node to be erased. This object will be of type TS_Node. The two individual lines under the nodes heading represent the two different aspects of this object: the base-class (Node) portion and the derived class (TS_Node) portion. We again separate these two aspects to emphasize how the code in TS_Node and Node interact. The next two lines similarly represent the j + 1st edge of the edges vector of the diagram editor.
The final line represents the **Scene sc**. In the sequence diagram, the numbered circles show special points of interest during execution. The grey circles are at points within the trace-saving classes, and the circles whose labels end with “t” represent places where the trace information is stored or modified.

![Extended sequence diagram for test_eraseNode()](image)

Figure 3.18: Extended sequence diagram for `test_eraseNode()`

To check that `eraseNode()` satisfies its interaction specification, we first must create an appropriate instance of `Test_DiagEditor` as a test case, and also choose the appropriate integer value to pass via the parameter `nd`. As described before, objects of type `TS_Node` and `TS_Edge` will be registered with the `this` object when it is set up. To initiate the test, which is represented by the solid arrow at the top-left of the figure, we invoke `test_eraseNode()` on `this` with `nd` as the parameter value. The method starts by checking the precondition—this is represented by the point labeled with a diamond with a single question mark inside. Then, after the initial state of the
variables is saved, it pushes a new trace \texttt{tau} on the \texttt{TraceStack} at the point labeled (1t). Next, \texttt{eraseNode()} is called on the object.

Consider what happens when \texttt{eraseNode} executes. First, it invokes the hook method \texttt{unDraw()}, which we have overridden in \texttt{TS\_Node}. Since the \texttt{Nodes} of the diagram object that \texttt{eraseNode()} was applied to are of type \texttt{TS\_Node}, this call is dispatched to the implementation of \texttt{unDraw()} in \texttt{TS\_Node}. This dispatch is represented by the solid arrow labeled \texttt{unDraw} from the lifeline of \texttt{this\_DiagEditor} to the vertical line labeled \texttt{TS\_Node} under the heading \texttt{nodes\_nd}. Now \texttt{TS\_Node\_unDraw()} will delegate the call to \texttt{Node\_unDraw()}, which is represented by the next arrow from \texttt{TS\_Node} to \texttt{Node} under the \texttt{nodes\_nd} heading. But, before it does so, it records appropriate information about this call on a trace record variable \texttt{caller}, initiates a new trace, and pushes it on \texttt{TraceStack}. These actions take place at label (2t) in the figure. The code for \texttt{unDraw()} found in the \texttt{Node} class will be responsible for updating the \texttt{Scene}, and should require some interaction between the code at label (3) and the \texttt{Scene} object \texttt{sc}. This interaction, which may entail several calls to the \texttt{Scene}, is represented by the single arrow from the \texttt{Node} lifeline to the \texttt{sc\_Scene} lifeline right after the (3) label\textsuperscript{16}. After \texttt{Node\_unDraw()} completes execution and returns, shown by the dotted back-arrow, we are back at label (4t). At this point, control is at \texttt{TS\_Node\_unDraw()} which now pops the previous trace off of \texttt{TraceStack} (the trace associated with \texttt{unDraw()}’s execution), records appropriate additional information on the \texttt{caller} trace record, and then appends this record to the topmost trace of \texttt{TraceStack} (the trace associated with \texttt{eraseNode()}’s execution).

\textsuperscript{16}It is worth noting that if these interactions with \texttt{Scene} involved hook calls, these calls would be recorded on the trace associated with the execution of \texttt{unDraw()}—the trace that just we pushed onto the \texttt{TraceStack} at label (2t).
After \( \text{TS}\_\text{Node.unDraw()} \) finishes, control returns to the body of \( \text{eraseNode()} \). Now, \( \text{eraseNode()} \) looks for the edges that were adjacent to the erased node, and proceeds to erase them. Here, we assumed that \( \text{edges}_j \) was adjacent to \( \text{nodes}_n \) for this execution run, so now \( \text{eraseNode()} \) invokes \( \text{unDraw} \) on this edge. But, since it is of type \( \text{TS}\_\text{Edge} \), the call is dispatched to \( \text{TS}\_\text{Edge.unDraw()} \) and not \( \text{Edge.unDraw()} \). This call is represented by the arrow labeled \( \text{unDraw} \) from \( \text{this:DiagEditor} \) to \( \text{TS}\_\text{Edge} \). For erasing an edge, the process of initiating a new trace and pushing it on the \( \text{TraceStack} \) (label (5t)), delegating the call to the corresponding method in \( \text{Edge} \) (the next solid arrow labeled \( \text{unDraw} \)), its update of the \( \text{Scene} \) (label (6)), the subsequent return, and the saving of the results and appending the record to the current trace (label (7t)) is similar to before. After \( \text{TS}\_\text{Edge.unDraw()} \) returns, control is back at the \( \text{eraseNode()} \) method defined in the \( \text{DiagEditor} \) class. Since there are no more adjacent edges to erase, \( \text{eraseNode()} \) itself returns, and control is back at \( \text{Test.DiagEditor.test_eraseNode()} \).

The final action, the one that we have been building up towards, is to check if the preserves clause and postcondition specified in the trace-based specification for \( \text{eraseNode()} \)—with \( \text{tau} \) substituting for \( \tau \) and using the \( \_\text{pre} \)-variables in place of the \#-ed values of the diagram editor’s fields—is satisfied. This point is labeled by the diamond with the double question mark.

By defining \( \text{TS}\_\text{Node} \) and \( \text{TS}\_\text{Edge} \) as derived classes of \( \text{Node} \) and \( \text{Edge} \), and by overriding the hook methods in them, and by using these derived classes in constructing the diagram editor object to be used during monitoring, we are able to exploit polymorphism to intercept the calls made to the hook methods during execution of template methods such as \( \text{eraseNode()} \). These redefinitions allow us to record information about these calls and their returns without having to make any changes to
the base framework code—indeed without having any access to the source code of the framework. This allows us to achieve the goal of checking the grey-box behavior specified in traces of template methods.

This methodology is applicable to OO frameworks in general, where we would like to check template methods against their trace-based specifications. By “plugging-in” suitable trace-saving classes in place of the derived classes that would normally be built by application developers, we can intercept all of the necessary calls made to hook methods. If a method call cannot be intercepted in this way, then this call would necessarily be made to a method body implemented in the original framework code. Because this call is internal relative to the framework (as opposed to external calls, which are dispatched to the application code) this behavior should already be accounted for in the specification, and the fact that such an internal call is made should be opaque to those using the framework component. In this way, the trace-based specification can be viewed as a description of how the framework component interacts with the application code.

3.3.3 Running TCGen and Sample Output

The TCGen tool is written in Java, using a tokenizer that was generated using Java-Lex. As input, TCGen takes files containing trace-based specifications written in a lightweight variant of JML, and allows for the specification of traces, and the flagging of the hook methods to be tracked. An alternate approach would be to designate all non-final methods as trackable, but we decided to allow the user greater flexibility in choosing exactly which enrichable methods to intercept. This can be of help in debugging complex frameworks by omitting, from the trace, calls to hook
methods that the user is not currently interested in, and allows him or her to focus on specific parts of the framework.

Given a specification file for a class containing hook methods, TCGen first generates specialized trace record classes for each of the methods flagged as a hook using its stated signature. As mentioned before, these classes use auxClone() methods to copy object state into the trace record, which are themselves generated along with the auxEquals() methods in a Util class. Then, trace-saving versions of the given classes are generated that contain the redefined hook methods described in Sec. 3.3.1. In addition to the trace saving capabilities coded into these redefined hooks, if there are any pre- or postconditions, ensures, or signals clauses specified for that hook method, the proper checking code will also be inserted into the implementation as previously described. This helps pinpoint problems while monitoring template methods if there is something wrong with a subsequent call to a hook method. This assertion code generated in the redefined hooks is produced in the same way as it is within the test methods.

Now, for a class that we want to use as a testing class as described in Sec. 3.3.1, we also input a similar specification file into TCGen. Those methods for whom a requires, preserves, ensures, or signals clause has been specified, a test method is generated. It is entirely possible that this class can contain both template and hook methods, such as the Order class from the previous chapter. In such cases, the trace-saving redefinitions of the hook methods are given in the testing class; recall that testing classes extend the original framework class, and thus can contain these redefined hooks. Essentially, the testing class and the trace-saving class are one and the same in this case. At present, TCGen does not generate test cases in the test class, but
does create skeleton calls to the test methods in \texttt{main()}, where the user is required to construct test values by hand.

After the trace-saving, trace record, and testing classes have been generated by TCGen, and the user has made the necessary modifications to them, these classes can be compiled, and the test class executed to initiate the actual testing. An example of the system’s output from a sample test run for \texttt{moveNode()} is shown in Fig. 3.19. To help illustrate what is going on in this example, the \texttt{show()} method for the \texttt{Scene} class, which prints out its contents as text, is invoked on the \texttt{DiagEditor} test case’s \texttt{sc} field on three occasions during testing (the output lines starting with \texttt{SCENE:}).

Before we invoke the \texttt{test\_moveNode()} method, a \texttt{TS\_Node} and a \texttt{TS\_Edge} object are registered with the test case object, and it is built up to a suitable test case by adding nodes and edges to the diagram. The first three lines (in all caps) are output by the framework, indicating that it being initialized, and is ready to be used. Next, in the testing code, we add nodes with center point coordinates of \((1,2)\) and \((5,6)\), and an edge connecting them by invoking \texttt{addNode()} and \texttt{addEdge()}. Because the node and edge prototypes registered with the test case object are of types \texttt{TS\_Node} and a \texttt{TS\_Edge}, the creation of the nodes and edge result in calls to the \texttt{clone()} and \texttt{draw()} hook methods, which are intercepted by the trace-saving classes. The next six lines of output are produced by the redefined hooks, to indicate that they have been called. Then, we invoke \texttt{show()} on the \texttt{Scene}, and get the next line of output showing us its contents in text form. We then add another node, with center \((7,3)\), and another edge, from nodes with centers \((7,3)\) to \((5,6)\). Then, we again invoke \texttt{show()}, before we run the test.
REGISTERING NODE
REGISTERING EDGE
REGISTRATION COMPLETE, DIAGRAM EDITOR READY.
  Method Node.clone called.
  Method Node.draw called.
  Method Node.clone called.
  Method Node.draw called.
  Method Edge.clone called.
  Method Edge.draw called.
SCENE: Node at (1,2), Node at (5,6), Edge from (5,6) to (1,2)
  Method Node.clone called.
  Method Node.draw called.
  Method Edge.clone called.
  Method Edge.draw called.
SCENE: Node at (1,2), Node at (5,6), Edge from (5,6) to (1,2), Node at (7,3),
Edge from (7,3) to (5,6)
  Method Node.clone called.
  Method Node.draw called.
  Method Edge.clone called.
  Method Edge.draw called.
  Method Edge.draw called.
Postcondition of moveNode not met!
tau = < ("draw(Scene)", {{1,2}}, Scene@18fe7c3);
  ("draw(Scene)", {{-5,-6}}, Scene@18fe7c3);
  ("draw(Scene)", {{7,3}}, Scene@18fe7c3);
  ("draw(Scene)", {{{-5,-6}}, {{1,2}}}, Scene@18fe7c3);
  ("draw(Scene)", {{{7,3}}, {{-5,-6}}}, Scene@18fe7c3)>
Test number 1 failed!
SCENE: Node at (1,2), Node at (-5,-6), Node at (7,3), Edge from (-5,-6) to (1,2),
Edge from (7,3) to (-5,-6)

** Test Results **

Number of tests run: 1
Number of tests successful: 0

Figure 3.19: Output from sample run
Now, we invoke test\_moveNode() on this diagram editor, indicating via the parameters to move the node at index 1 (the node centered at point (5,6)), to the point (−5, −6). The test invocation message is produced by the testing code, and indicated in the output. The next five indented lines in the output show the hook method calls made by moveNode(), and the next line alerts us that at the end of execution, moveNode()’s postcondition was violated. When a postcondition violation occurs, the TCGen generated test class automatically displays the trace to the screen. For each TraceRec on the trace, the default action is to display the method name, the target and parameter objects’ hash codes (which act as a representative for references), their pre- and post-state representations (given by their toString() values), and the hash code and post-state of the result if the method returns a value. To represent state, the toString() method, which is common to all Java objects, is redefined in each of the trace-saving classes by TCGen to show the toStringed values of each of its fields, giving us at least a “one-deep” view of those objects. To make the trace easier to read for this example, however, we simplified the trace record output to only include the method name, the target’s pre-state, and the parameters’ hash value.

Looking at the trace, we see that draw() was invoked on every node and edge of the diagram, which is not what should happen according to the trace-based specification of moveNode() in Fig. 3.9. It states that first unDraw() should be called on the given Node object and its adjacent Edges, and then after the center of the node is adjusted, draw() will be called on all of these objects again. Here though, by examining the trace, and the resulting Scene whose contents are displayed after the test failure notice in the output, we might conclude that instead, clear() was first invoked on the Scene, and then all of the elements of the diagram were re-rendered after the point was
moved. (Since clear() is not a hook, such a call would not appear on the trace.) In any case, the trace-based postcondition was clearly violated, and the testing apparatus detected this violation, even though its standard black-box behavior was as expected.

3.4 Related Work

Others have previously examined many of the problems we have discussed in dealing with monitoring and testing of OO systems. For example, Perry and Kaiser [67], and Smith and Robson [75] mention the difficulties that inheritance can cause when inherited base class code does not cooperate with redefined methods in a derived class, and that suitable test cases appropriate to the derived class will have to be designed. In the work of Harrold, McGregor, and Fitzpatrick [38], which we mentioned in the previous chapter, an algorithm is proposed that identifies which tests of the base class are applicable to the derived class. Their main point is that if a particular method m() in a base class is neither redefined in the derived class, nor is it dependent (either directly or indirectly) on other methods that have been redefined in the derived class, then tests used to validate m() in the base class can continue to be used in the derived class. Hsia, Li, Kung, Hsu, Li, Toyoshima, and Chen’s [41] work is similar, but extends this idea to include situations where changes are made in already existing classes. None of these authors deal with specification-based monitoring and testing, however, and are more concerned with identifying suitable test cases based on the internal structure of the classes.

Other authors have addressed similar concerns related to testing of polymorphic interactions [57, 9, 1, 71]. In most of this work, the approach is to test a template method t() by using objects of many different derived classes to check whether t()
behaves appropriately in each case, given the different hook method definitions to which its calls are dispatched, depending on the derived class that the object is an instance of. The work of Wu, Chen, and Offutt [84] is similar, but they do go into some detail about the problems of testing OO components in the absence of source code. As the others, though, they do not provide any methodology to test against grey-box specifications. Their approach is simply to specify the internal behaviors using enriched UML diagrams, and use those specification to come up with test cases which will be used to check against standard black-box specifications. Our focus is not on test case selection, but rather how we can achieve runtime monitoring of framework code independently of the derived classes, and how it can be done without source code. Without such monitoring techniques, we cannot test specifications that contain grey-box information in the first place when the source code is unavailable.

The work done by Cheon and Leavens’s [20] is similar to our work in that it focuses on specification-based monitoring, and not on test case selection. In their system, given a class and its specification, a test class is generated that allows for checking the pre- and postconditions of any method of the class that is invoked. Although this allows for the pre- and postconditions of hook methods $h1()$ and $h2()$ to be checked during testing of $t()$ as in our methodology, they do not consider specifications which can include grey-box information such as traces, and so do not deal with saving trace information. Their approach is also different in that they use a special compiler which inserts pre- and postcondition checks at the appropriate places in the compiled code, which requires the source code of the classes to be monitored.
An approach that uses special classes that extend the components to be monitored for black-box monitoring has been developed by Edwards [28], and by Gross, Scheiferdecker, and Din [36], independently. As in our approach, they use special “wrapper” classes that contain the actual code to do the pre- and postcondition checking. However, instead of defining a monitoring class TC to be a derived class of C, which is our approach to creating trace-saving classes, their wrapper class TC *encapsulates* an object of type C. The class TC implements the same interface as C, allowing for it to be used polymorphically in place of C, and delegates the calls to the encapsulated object while implementing the pre- and postcondition checking code. As in our approach, the source code of C is not needed to do this. However, their work does not consider grey-box specifications, and the approach cannot support monitoring of hook method calls in some situations. Here, once a call is delegated to the encapsulated object, we have no control over what happens—including saving trace information when hooks are called—until the call has been completed. This problem manifests itself in situations where a hook is called by a template method that is in the same class, such as where `setUp()` in the `DiagEditor` class calls `registerAll()`. Because the call to the template method `setUp()` would be delegated to the encapsulated `DiagEditor` object, the “wrapped” version of `registerAll()` would not be called.

The idea of using special extension classes to provide monitoring code is found in a few other places in the literature. Meyer [60] alludes to using derived classes in this way, but does not go into any detail, and only talks of black-box specifications. Weide [83] uses derived classes for precondition checking, but does not check postconditions (let alone grey-box postconditions), and these classes are coded by hand. To our knowledge, our approach is unique in that it has tool support for the generation of
such classes from trace-based specifications, and allows for the monitoring of trace information.
MONITORING CONTROL-FLOW THROUGH INHERITANCE HIERARCHIES

As we have seen, polymorphism enables designers to modify and enhance the behaviors of OO systems by redefining methods in their implementation of derived classes. Not only will the redefined methods exhibit new behavior, but polymorphism ensures that other methods that are not redefined will also exhibit new behavior whenever these methods are called on objects of the derived classes. With this power and flexibility, we also get a higher degree of complexity when trying to reason about and better understand OO systems. In Chapters 2 and 3, we discussed how using traces in specifications and monitoring can provide a solution to these problems by capturing critical behaviors: what enrichable methods are called in what sequence, under what conditions, etc.

In this chapter, we look at another aspect of this complexity caused by polymorphism and other OO features, which has been dubbed the *yo-yo problem* in the literature [82, 9, 63]. The yo-yo problem is concerned with the way that control flows between different methods’ code defined in an OO program, including code found at different levels of inheritance hierarchies. While a trace tells us how a single method interacts with parts of the system that can change, they do not characterize chains of
interactions, or how different parts of the code are dependent on other parts. Runtime monitoring of these interaction chains would not only give us a better understanding of “who calls who”, but when coupled with pre- and postcondition checking of the methods invoked, we get a way to help find the source of possible bugs. In this chapter we present just such a monitoring technique; a technique that exploits polymorphism in much the same way as we did in Chapter 3.

4.1 The Yo-Yo Problem

There are two distinct aspects that contribute to the complexity of control-flow among the methods defined in different classes of a program. The first involves the related mechanisms of inheritance and dynamic binding, which we will illustrate in the following example. Suppose we have four classes, B4 a derived class of B3, B3 a derived class of B2, and B2 a derived class of B1. A method a() is defined in B1, which is inherited by B2, then redefined in B3, and then inherited by B4. Another method b(), defined in B1 and inherited by the other three classes, contains in its body a call to a(). When this b() is applied to an object that is an instance of B4, dynamic binding will ensure that the call to a() that is made from within its body will be dispatched to the a() defined in B3. On the other hand, if this a() were to be applied to an object that is an instance of B2, the same call will go to the a() defined in B1. Although b() is inherited by the various derived classes, what it actually does—in particular, what methods it invokes as it executes—depends critically on which particular class the object in question is an instance of. While such features make OO very flexible, it makes following the control-flow rather difficult.
The second aspect that contributes to the complexity of control-flow is, somewhat paradoxically, one designed to avoid the dynamic binding: the super-call mechanism. For instance, one of the methods of B3, say a(), may contain a call such as super.a(), which will then be dispatched to the version of a() defined in B1 (which was inherited by B2). The reason that standard OO languages provide the super mechanism is that often the redefinition of a method such as a() in a derived class has to perform all of the tasks carried out by the base class definition of the method, plus some additional activities typically related to the additional state (in the form of new member variables) of the derived class. While the former task could be achieved by duplicating the code of a()’s definition from the base class, it would clearly be simpler if we could invoke the base class a(), as we can with the call super.a(). But this means that control transfers to the base class definition of the method—which was supposedly superseded by the derived class definition—which might then invoke other methods that will be dynamically dispatched, unless those invocations also use the super mechanism.

In our discussion, we will use the term *up-call* to refer to calls using the super mechanism since such a call will result in control going from the current method to a method defined in an ancestor class. Similarly, we will use the term *down-call* to refer to calls that are dispatched based on the class that the object on which the method is applied is an instance of. We should perhaps note that “down-call” is not always an accurate description; if a method m() defined in B4 is invoked on a B4 object, and the body of m() contains a call to a(), control will flow from B4.m() up to B3.a(), since B4 inherited a() from B3. In contrast, any super-call necessarily results in control flowing up.
The term “yo-yo problem”, which was coined by Taenzer, Ganti, and Podar [82], was intended to convey the effect of dynamically dispatched calls that typically transfer control to methods defined in derived classes (classes that are lower down in the inheritance hierarchy), alternating with calls using the `super` mechanism that transfer control to methods in ancestor classes (classes that are higher up in the inheritance hierarchy). In their words, “The combination of polymorphism and method refinement (methods which use inherited behavior) make it very difficult to understand the behavior of the lower level classes and how they work.” Since then, several authors have commented on the yo-yo problem, and have noted that it is one of the key components contributing to the unique difficulties that OO system designers and analysts face. Binder [9], for example, lists “[T]he loss of intellectual control that results from spaghetti polymorphism (the yo-yo problem)” as one of the unique bug hazards of the OO approach.

Given the potentially complex control-flow among the methods of the various classes, it would clearly be useful to develop an approach that enables us to track this flow automatically, and a way to depict it graphically. Binder [9] presents a hand-drawn representation of this, which we have reproduced in pseudo-UML format in Fig. 4.1. We will refer to such graphs as *yo-yo graphs*. At the left margin of the graph is a diagram of the (static) inheritance hierarchy of the classes, where the methods are specified as being inherited (labeled with `inh`), as making a `super`-call to the same method (labeled `ref` for “refinement”), or as being defined in the class without making a `super`-call to the same method (listed with no label). The main part of the graph shows the control-flow among the various methods of the various classes that results when a particular method is applied to an instance of one of the classes.
Figure 4.1: Example yo-yo graph
Interestingly, the hand-drawn yo-yo graph from [9] seems to be incorrect and possibly inconsistent with the accompanying description of the program being considered. Here, the method $a()$ is invoked on an object of class $C_5$. The first problem we see is that the method $d()$ is shown as being defined in class $C_3$, and not inherited by $C_3$. This is not consistent with the arrow going from the $C_5.d$ oval to the $C_2.d$ oval, which is supposed to represent the super-call made by $C_5.d()$. ($C_5.d()$ is labeled as a refinement, and so must call super.$d()$.) If such a call was made, control would have to pass through the method $d()$ defined in class $C_3$. To add to the confusion, the text description of the program found in one of the tables states that $C_5.d()$ invokes super.$a()$, and not super.$d()$. Similar inconsistencies are seen in the example with the method $c()$, which is shown as defined rather than inherited in class $C_3$ in the graph, and the statement in the text that $C_4.c()$ makes a call to super.$a()$ and not super.$c()$.

While these inconsistencies may be purely a typographical error, discovering that there is an inconsistency takes some effort for even such a simple program, attesting to the complexity of analyzing the flow in such programs. Any mechanical assistance that system designers can utilize in tracking the control-flow would be valuable. In this chapter, we present PolyTracker, a prototype tool that we have implemented that exploits polymorphism to track the control-flow, and produces the yo-yo graph automatically.

### 4.2 Methodology Overview

Let us first consider the down-calls. Corresponding to a class $C_i$ in an OO program whose control-flow we are interested in tracking, we will introduce a derived class $T_{C_i}$ (the leading “$T$” stands for “tracking”) and redefine the methods of $C_i$ in $T_{C_i}$. If we
are interested in seeing what control-flow would result when a method `a()` is applied to an object that is an instance of, say, `C5`, we create an object of type `TC5` and apply `a()` to it. As we will see, a `TCi` class is defined in such a way that the resulting control-flow will be essentially the same as if we had applied `a()` to an instance of `C5`. The only difference is that the down-calls will be first intercepted by the methods we define in `TC5`, which will record suitable information about the call, and then forward the call to the actual method that would have received the call if the object had been an instance of `C5`. This can be done without making any changes to the original classes `Ci`, and indeed without having any access to the source code of those classes. This approach is very similar to that used by TCGen described in the previous chapter, where trace-saving classes are generated to intercept the calls to hook-methods.

The up-calls present a more difficult challenge. The problem is that when a call such as `super.d()` is made from within the body of, say, `C5.d()`, control flows up to the `d()` defined in the closest ancestor of `C5` independent of the class that the current object is an instance of. In other words, there is no way to use polymorphism to intercept such a call. Therefore, we will use a slightly more involved approach to handle these calls: In addition to defining methods in `TCi` that will intercept the down-calls, it will also make certain minor modifications to `Ci` and the classes above it in the inheritance hierarchy to facilitate recording information about the up-calls. With the `TCi` class and the necessary changes in place, both the down- and up-calls can be captured and recorded during execution. With this information in hand, the PolyTracker tool we have implemented constructs and displays the yo-yo graph, as well as information about the values of arguments passed during the various method calls, and the results returned.
One important point to note is that although some minor modifications have to be made to Ci and the classes above it in the hierarchy, the modified program behaves in exactly the same manner as the original program does with respect to the objects that are instances of these classes. That is, it is only when we use an instance of the TC\textit{i} class that we introduce, that the new behavior in the form of recording information about the up- and down-calls comes into play. This approach is quite different from the instrumentation techniques that are occasionally used, where special debugging code is inserted into the source code to help system developers observe what is happening during runtime. Using our technique, the system methods’ control-flow can be monitored and analyzed by compiling the TC\textit{i} class in with the regular classes, and applying the methods to TCi objects which generate the yo-yo graph as they execute. Once we are satisfied with Ci’s behavior, we can put aside the TCi class, and ship the system’s class code without making further changes\textsuperscript{17}. The modified system files will compile without the TCi class, and will behave the same as they did originally.

4.3 Tracking the Method Calls

Consider the program shown in Fig. 4.2 consisting of classes C\textit{1} through C\textit{5}, with each class (except C\textit{1}) being a derived class of the one immediately above it. This is a fleshed-out version of the example used in [82] and [9], where we provide code for the various methods in these classes. However, these methods are not really intended to do anything particularly interesting; our focus rather is on how control flows among the various methods as a result of the use of polymorphism and calls to \texttt{super} methods.

\textsuperscript{17}Or, if we do not want to make the source code available, we can recompile the class code without TCi, and ship the resulting object files as-is.
abstract class C1 {
    protected int x = 0;
    protected void a() { x++; x = b(); c(x-1); }
    abstract protected int b();
    abstract protected void c(int k);
}

class C2 extends C1 {
    protected int y = 0;
    protected int b() { y = 2*x; int j = d(y); return y+j; }
    protected void c(int k) { x = x - k; }
    protected int d(int k) { c(k+1); return x; }
}

class C3 extends C2 {
    protected boolean p;
    protected void a() { p = !p; super.a(); }
    protected int b() { return super.b(); }
}

class C4 extends C3 {
    protected int z;
    protected void a() { super.a(); z++; }
    protected void c(int k) { super.c(k); z=z+x; }
}

class C5 extends C4 {
    protected boolean q = true;
    protected int d(int k) { q = !q; return super.d(k); }
}

Figure 4.2: Program to be tracked (original source code)
As in [82] and [9], our discussion will focus on tracking the \texttt{a()} method when invoked on an object of class \texttt{C5}. For this invocation, since the closest ancestor of \texttt{C5} that has a definition of \texttt{a()} is \texttt{C4}, it is that definition that will be executed initially. That method definition invokes \texttt{super.a()} which calls \texttt{C3.a()}, which in turn also invokes \texttt{super.a()} which calls \texttt{C1.a()}. That method invokes \texttt{b()} and then \texttt{c()} and these calls will be dispatched to their respective definitions applicable to instances of \texttt{C5}, which are \texttt{C3.b()} and \texttt{C4.c()}, respectively.

4.3.1 Down-calls

Consider a call such as \texttt{m()} that appears in the body of some method \texttt{n()} in some class \texttt{Cj}. Suppose the this object on which \texttt{m()} is being invoked is an instance of \texttt{Ci}. When this call is executed, it will be dispatched to the definition of \texttt{m()} that is in \texttt{Ci} or, if \texttt{m()} is not (re-)defined in \texttt{Ci}, the one in the closest ancestor of \texttt{Ci} that has such a definition.

In order to intercept such calls, we will create a derived class \texttt{TCi} corresponding to the \texttt{Ci} class in the original program. In \texttt{TCi}, every method that is applicable to objects of type \texttt{Ci} will be redefined\textsuperscript{18}. Consider, for example, the class \texttt{TC5} corresponding to the class \texttt{C5} that appears in Fig. 4.3. The methods applicable to \texttt{C5} objects are \texttt{a()}, \texttt{b()}, \texttt{c()}, and \texttt{d()}. We have redefined each of these in \texttt{TC5} such that information about the call and its return—including information about the object, parameters, and possible return values—are saved by the singleton \texttt{Tracker} object which logs each of these calls and returns. This is similar to the technique used in the previous chapter, where trace information was stored before and after the the method of the

\textsuperscript{18}More precisely, we should say every non-final method will be overridden since final methods cannot, of course, be overridden.
class TC5 extends C5 {
  protected void a() {
    Tracker.record(new CallRec("a", this.toString(), ","));
    super.a();
    Tracker.record(new RetRec("a", this.toString(), ",");
  }

  protected int b() {
    Tracker.record(new CallRec("b", this.toString(), ","));
    int _ret = super.b();
    Tracker.record(new RetRec("b", this.toString(), "result=\"\"+\"_\"\"\"+\"\"\"\"+\"\"\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\"+\"\”}
parent class (in this case, C5) was invoked using the super mechanism. Analogous to the TraceRecs that save a single hook call on a Trace, each of the calls and returns here are recorded in a TrackerRec object which are saved in the Tracker object’s log variable (see Fig. 4.4). The start of a (regular) call is recorded with a CallRec, and the return is recorded with a RetRec; both of these are derived classes of TrackerRec.

We will discuss the SuperRec classes in the next section.

Along with the name of the invoked method, a TrackerRec also records the target object, parameter, and result state. However, unlike the TCGen methodology, specialized TrackerRec classes are not created. Such specialized classes are not needed here because state is stored in “stringified” form, as defined by the toString() method defined for the given classes, instead of saving state in clones of objects. Also, the state of the parameters and return values are saved as a single String, otherState. We chose to use this approach, instead of the former, to simplify the implementation, and to reduce the runtime and memory overhead associated with object creation and

Figure 4.4: Tracker and TrackerRec classes
cloning. Having String representations for state here is reasonable, since this state information is intended for human inspection, rather than evaluation in specifications.

With this scheme, if the method b() is invoked on an object o of type C5, the call will be dispatched to the definition in C3.b(), since C5 inherits rather than redefines b(). The method will execute normally without being affected by TC5, and will not be tracked. But now, suppose that o is an instance of TC5 instead. In this case, the call to b() will be dispatched to TC5.b(). That method will save information about this call and then invoke super.b(). That invocation will then be forwarded to C3.b(), since C5 is the base class of TC5 but does not override b(). Instead, it inherits it from two classes up in the hierarchy, at C3. Thus, the method that is called at this point is the same as the one that would be called on a regular C5 object.

While C3.b() is executing, it will invoke super.b(), which in turn will invoke d(), etc. We will discuss shortly how these methods will also be recorded on the Tracker object, but we will move forward to when C3.b() finishes executing. When C3.b() returns, control resumes in TC5.b(), which records that the call to b() has returned, and stores information about the current object state and result. Then, TC5.b() returns the same value that was returned by the call to C3.b(). In this way, the original call to b() on the TC5 object will yield the same result as it would had we invoked b() on a C5 object. The difference is that the information about the call to, and the return from, b() has been recorded.

4.3.2 Up-calls

Consider the call super.b() that appears in the definition of C3.b(). When this call is executed, control will immediately transfer to C2.b(), independent of the runtime
class C3 extends C2 {
    protected boolean p;
    protected void a() { p = !p; C3.super_a(); }
    protected int b() { return C3.super_b(); }
    protected void C3.super_a() { super.a(); }
    protected int C3.super_b() { return super.b(); }
}

Figure 4.5: Modified class C3

type of the object at hand. Thus, no matter what we do in the derived TCi classes, we cannot intercept this call. What we can do instead is rewrite these calls in such a manner that they can be intercepted.

Consider the modified class C3 in Fig. 4.5. This class differs from the original C3 in Fig.4.2 in two respects. First, we have introduced two new methods, C3.super_a() and C3.super_b(), each of which simply calls the corresponding super method. Second, each of the super-calls that appeared in the methods of the original C3 have been replaced by a call to the corresponding proxy method that we have introduced. We see that the call super.a() in the original C3.a() has been replaced by a call to C3.super.a(), and similarly for the call to super.b() that appears in the original C3.b(). Note that we have not introduced methods C3.super.c() or C3.super.d(); this is because there are no calls of the form super.c() or super.d() in any of the methods of the original C3. If such calls had existed, these methods should be defined.

With these changes, the modified C3 will still behave in exactly the same way as the original C3, as far as instances of C3 are concerned. To do the tracking of these calls, though, we need to add appropriate methods to TC5, and of course use TC5 objects during monitoring. In the TC5 class defined in Fig. 4.6, we have redefined
class TC5 extends C5 {
    ...a(), b(), c(), d() as in Fig. 4.3...

    protected void C3.super_a() {
        Tracker.record(new SuperCallRec(“C3”, “a”, this.toString(), “”));
        super.C3.super_a();
        Tracker.record(new SuperRetRec(“C3”, “a”, this.toString(), “”));
    }

    protected int C3.super_b() {
        Tracker.record(new SuperCallRec(“C3”, “b”, this.toString(), “”));
        int _ret = super.C3.super_b();
        Tracker.record(new SuperRetRec(“C3”, “b”, this.toString(), ”result=” + _ret));
        return _ret;
    }

    ...similar definitions for C4.super_a(), C4.super_c(), and C5.super_d();
    C1 and C2 do not make super-calls...
}

Figure 4.6: TC5 class code (final version)

the methods C3.super_a() and C3.super_b() in the same way as we redefined the original methods a(), b(), c(), and d() in Fig. 4.3. Thus, if we use an object that is an instance of TC5, rather than an instance of C5, and apply the method b() to it, the initial call will be dispatched to TC5.b(), which will record the down-call to C3.b() as described in Sec. 4.3.1. Now, the call to C3.b()—where the call to super.b() in the original source code of C3 has been replaced by the call to C3.super.b() in the modified C3 in Fig. 4.5—will invoke C3.super.b(). This call will be dispatched to the redefinition of C3.super.b() in TC5. This method, as the other tracking methods, first saves the information about this call on Tracker. For recording super-calls, however, we use SuperRec objects, which contain one more piece of information than the previous TrackerRecs we have seen: the class from which the super-call was made. After
recording the pre-state, TC5.C3.super_b() then forwards the call to C3.C3.super_b() using the super mechanism, which in turn forwards the call to the b() defined in C2, which was the method that was called from the original C3.b().

In addition to redefining in TC5 the super-call proxy methods from the class C3, we also need to provide similar redefinitions in TC5 for those proxy methods to be added in the other classes in the inheritance hierarchy. Upon inspection of the original code in Fig. 4.2, we see that methods in the classes C4 and C5 also make super-calls, namely a() and c() in C4, and d() in C5. Thus, to allow for tracking of those calls, we need to make similar modifications to those classes as we did for C3 in Fig. 4.5, and add the appropriate super-call proxy definitions to them, C4.C4.super_a(), C4.C4.super_c(), and C5.C5.super_d(). With these additions, TC5 will allow us to intercept all down- and up-calls invoked upon objects of type TC5, and save the information about these calls in the log in the Tracker object.

4.4 Running PolyTracker and Sample Output

Our prototype implementation operates in two distinct phases: the monitoring class generation phase, and then the yo-yo graph rendering phase. The first phase starts by inputting into PolyTracker the names of the classes and their methods that should be tracked. Then, the original source code of these classes are input into the tool. These classes are modified, where each of the super-calls to methods that are to be tracked are transformed into a call to the corresponding super-call proxy method of the class. The tool introduces the trivial definitions for these proxy methods in these classes as well.
In the second part of the first phase, PolyTracker produces the tracking class \( \text{TC}_i \) for the class of the specified target object \( C_i \). In our example, we were concerned with objects of type \( C_5 \), and so the tracking class \( \text{TC}_5 \) was generated. Note that if we also wanted to track calls for objects from the other concrete classes \( C_4 \), \( C_3 \), or \( C_2 \), we would have to generate a separate tracking class \( \text{TC}_4 \), \( \text{TC}_3 \), or \( \text{TC}_2 \). However, the super-call proxy modifications made to the original system classes will work properly within the context of any of these tracking classes; such modifications are independent of the target object’s class. We have seen that these tracking classes do not in any way depend upon the details of the methods defined in the original classes. All that is needed to define these classes are the method names and the parameter and result-type information defined in the original classes. With this information, the tracking classes are produced mechanically.

Once the system classes have been modified, and the tracing classes have been produced by the tool, these classes are compiled using the standard Java compiler. We then invoke the method to be tracked on a \( \text{TC}_i \) object. During the execution of this method, the calls and returns of the monitored methods will be recorded by the Tracker. When the method is finished, we are ready to enter phase two, and render the yo-yo graph.

The method \texttt{renderYoYo()} of the Tracker class takes the information stored in the \texttt{log}, as well as the class hierarchy information, and first calculates the dimensions of the boxes and sizes of the diagram elements so they can be properly displayed in a graphics window. After the class diagram is rendered at the left of this window, the calls and returns in the \texttt{log} are processed and displayed in the main part of the window, along with connecting arrows. An example of a yo-yo graph produced by
PolyTracker is shown in Fig. 4.7, which shows the results of invoking the method $a()$ on a $TC5$ object.

On the left side of the figure, we see the inheritance hierarchy. In the class $C2$, the method $a()$ is flagged as $inh$ to indicate that it is inherited from the base class $C1$. In $C3$, the same method is flagged as $rdf/ref$ to indicate that this method is redefined in this class, and that this redefinition is a refinement of the base class definition of $a()$ (i.e., $super.a()$ is invoked). We should note that in this example, an implemented method is either defined from scratch (as in the case of $C2.d()$), or, as in the case of $C3.a()$, invokes the $super$ class version of itself as part of its body; it does not make
a **super-call** on a different method. Methods that simply redefine a base class version of itself, but do not make a **super-call** would only carry the label **rdf**. Methods that make a **super-call** to another method are labeled **sup**, along with any other pertinent labels.

There are a few notable differences between the yo-yo graphs that we generate with the tool, and yo-yo graphs such as the one pictured in Fig. 4.1 found in the earlier literature. One simple difference is that we use arrows with short dashes to denote up-calls and arrows with longer dashes to denote down-calls, to distinguish between the two. Another difference is that we do not create *call nodes*—the boxes with rounded corners representing the method definition executed for each invocation—for those class methods that are inherited. The rationale for this is that if a class $C_j$ inherits a method $m()$, then there is no code for $m()$ in $C_j$ to be executed; such invocations “pass through” the class $C_j$ to its first ancestor class that does implement $m()$. The earlier graphs do use such call nodes for inherited methods in the bottom-most class in the hierarchy to draw attention to the fact that when the dispatching mechanism searches for an implementation of $m()$ to execute, it necessarily starts at this bottom class. Hence, in Fig. 4.1, we have the bottom nodes labeled $C_5.a$, $C_5.b$, and $C_5.c$. These nodes are not part of our graph in Fig. 4.7, and are not shown in the later literature, such as [63].

A much more noticeable difference between our yo-yo graphs and the others from the literature is that in addition to the flow of control at the time of the call being represented, we also picture the flow of control *as it returns* to the calling method. Control returning to a calling method is illustrated by the grey nodes and arrows in our graph, which can be distinguished from the white nodes outlined in black that
represent control resulting from the initial method invocation. The advantage of using such return nodes is that we can show the control-flow through the complete execution of a method that makes multiple calls, and accurately depict the dependencies between the calling and the called methods.

This advantage can be seen when we look closer at the traditional yo-yo graph in Fig. 4.1, alongside the associated program code in Fig. 4.2. In the code, we see that C1.a() makes two calls: one to b(), and one to c(). This first call is shown in the graph by the arrow starting from the node C1.a, found at the top of the diagram, and going down to the node C5.b. The second call, however, and the subsequent calls resulting from it are not shown. By following along in the graph while hand executing the code, we see that the call to c() shown in the graph is initiated from C2.d, not C1.a; this call was made as a consequence of C1.a()’s first call to b(), and is not related to the second call that it should make. The graph does not depict what happens after C1.a()’s first call to b() ends, and so does not go through what happens when it invokes c(). If we were to represent this call to c() by C1.a by simply linking the last call node visited (in this case, C2.c) to the node representing this call, it may appear that C2.c() is itself making a call to c(), which it is not. Chaining the call nodes together by chronology in this way is of limited use from the standpoint of program understanding, and especially debugging, since the notion of dependencies among method bodies would be lost. Instead, in our yo-yo graphs, we can explicitly show how control returns to C1.a() from its call to b(), and then show what happens when it invokes c().

In addition to displaying the yo-yo graph, PolyTracker allows us to output the information stored in the log variable to a file using the Tracker.storeLog() method.
This information, along with the produced yo-yo graph, can aid us in understanding the system’s behavior—whether it is functional or calling behavior—by allowing us to see the state of the system through each step of the graph.

4.5 Discussion

In this chapter, we have developed an approach that can be used to automatically track the flow of control among methods in an OO program. As we have mentioned in previous chapters, polymorphism contributes to the complexity of OO programs, essentially because it allows for different levels of classes in inheritance hierarchies to interact with one another, and gives us a means to enhance or change a system’s behavior by further extending these classes. The focus before, was on how we can specify and monitor a method’s behavior so we can modularly reason about their classes, and compose larger systems from these classes that will behave in a predictable way. Here, we are more concerned with the system as a whole, how individual parts interact with one another, and what parts are dependent on one another.

In monitoring the control-flow resulting from dynamic dispatch of method calls, we were able to exploit polymorphism by redefining the methods of a class Ci in a tracking class TCi which extends it. Then if the original methods were invoked on an object of type TCi, the resulting method call would be dispatched to the redefinition in TCi which then recorded the necessary information about the call, and then used the super mechanism to invoke the original method that would have been invoked. This did not require any change in the original classes of the OO program, indeed did not even require the source code of those classes. This is essentially the same
approach we used in Chapter 3, where we tracked polymorphic calls in the context of evaluating trace-based specifications.

Tracing the super-calls here was more involved. Such calls are not dynamically dispatched, and so we cannot use polymorphism directly to intercept these calls. Instead, we had to allow for slight changes to the source code which essentially exposed these call points in the code to polymorphism. By doing so, we could also capture and record the super-calls in the tracking class TCi, as we did for the down-calls.

The yo-yo problem itself is limited in focus, in that it is only concerned with the control-flow relative to a single object, and does not specifically deal with multiple objects and their interactions. However, as we have demonstrated in Chapter 3, interactions between different objects can be captured in situations where the system objects’ classes can be specialized in tracking/trace-saving classes. For recording all of the calls and returns, instead of using a TraceStack where some of the call information is discarded with each Trace that is popped off of it, a singleton Tracker object like the one we discussed in this chapter could be used, which should also record the target objects’ identity with each TrackerRec. Of course, a simple yo-yo graph would be insufficient to illustrate these interactions—what would be more appropriate would be an extended sequence diagram, such as the one pictured in Fig. 3.18 from the previous chapter.

There is a limitation to this approach, however. Although we are free to specialize Ci classes to TCi classes, it may not be possible to change the binding from a Ci object to a TCi object at some of the variable locations within the system. Frameworks rely heavily on this ability to change the objects bound to particular variables, so that they can be customized to fit the specific needs of developers. However, this is not
always the case for general software systems. One possible way to work around this
difficulty is to modify the source code of the system, so that each mention of the
class \( C_i \) in the code is replaced with \( T_C_i \). In this way, the tracking objects will
automatically be inside of the system. When it is determined that the system has
been adequately tested, either the occurrences of \( T_C_i \) can be changed back to \( C_i \), or, a
trivial alternative definition for \( T_C_i \) can be used and compiled with the system. This
trivial definition would add no state, inherit all of its methods from \( C_i \), and simply
invoke \( C_i \)'s constructors within its own constructors. Another possible option is to use
aspects from aspect-oriented programming to intercept the method calls—it is this
approach that we will use in Chapter 6, when we monitor design pattern contracts.

Before we move on in our discussion, it should be acknowledged that some authors,
for example, [62, 76, 81], have argued that given the complexity of the control-flow
resulting from the use of polymorphism and the \texttt{super} mechanism, that the use of
inheritance and polymorphism based on dynamic binding should be minimized or
avoided as much as possible. On the other hand, several authors, for example [60, 29,
30], provide convincing arguments for, and compelling examples that demonstrate the
power of, these mechanisms in building complex systems. Indeed, such mechanisms
are what sets OO languages apart from other module-based languages.

4.6 Related Work

The idea of producing graphs to represent control-flow in OO programs has been
investigated by several authors. For example, Lange and Nakamura [52, 53] present
a technique for tracing the execution of an OO program. Their technique is based on
accessing, at the machine level, specific information contained in the run-time structures as the program executes. Such an approach is specific to not just the language, but also the particular implementation. However, their approach has that advantage in that they can extract considerably more information about the program’s execution. They also discuss various graphical ways to display the information, such as interaction charts that indicate on each object’s lifeline the invocations the particular object makes.

De Pauw, Helm, Kimelman, and Vlissides [22] present an approach to visualizing the execution of OO programs, including object construction, destruction, and calls to methods. Their technique requires insertion of substantial amounts of code in the individual classes, which then have to be removed before the system is deployed. Their approach also depends on the RTTI mechanism of C++ to access, at run-time, information about the actual types of given objects. Jerding [42], as well as Sharp and Rountev [73], discuss ways in which the execution of OO programs can be visualized and displayed graphically. Their main concern is on filtering and extracting the most relevant information from the large amounts of data that may be obtained about the program’s execution, so that what is displayed is easy to comprehend, and at the same time, useful. Their work does not center on the question of how to obtain this information, which is our main focus.
CHAPTER 5

SPECIFYING DESIGN PATTERN CONTRACTS

In the previous chapters, we have discussed some of the features of object-oriented programming that give developers a means to extend and enhance software systems, without having to rewrite large portions of system code. We have also seen how these features can make understanding and reasoning about such systems more difficult. To help developers overcome these difficulties, we have presented a modular specification language, as well as two complementary monitoring techniques for OO systems.

We now turn our focus to a more recent development in OO software engineering—the use of design patterns. Patterns have proven valuable by providing developers with descriptions of time-tested solutions to common problems, as well as a common design vocabulary. However, design pattern descriptions are imprecise, and can lead to misunderstandings between developers working on the same system. To address this problem, we will present a specification language in this chapter that captures pattern requirements precisely, while not sacrificing the flexibility inherent in the patterns’ informal descriptions. Later, in the next chapter, we will discuss how we can monitor systems to determine if these requirements are satisfied.
5.1 Introduction

Design patterns have become an important part of software practice [33, 18, 2, 43, 69]. Patterns capture the distilled wisdom of the design community by describing proven solutions to commonly occurring problems, while outlining some of the pitfalls that should be avoided when applying a particular pattern. They are usually collected together into publicly available pattern catalogs, some of which focus on a specific problem domain. Many pattern names have become part of the common lexicon of developers and designers, allowing them to communicate abstract design ideas. This ability to quickly understand the structure of a given system and why it behaves in particular ways is crucial for the rapid development and evolution of such systems in today’s software engineering environment.

However, these benefits are undercut by two important factors. First, as we have mentioned, the informal style in which patterns are expressed creates opportunities for ambiguity and misunderstandings among those involved in the design and development of a system. If team members have different interpretations of how a pattern is to be applied and implemented to meet a specific need in a system, incompatibilities among different parts of the system are likely to arise. Second, there is very little tool support for aiding developers in discovering behavioral errors in pattern implementations.

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

The second problem we mentioned—the lack of tool support for detecting pattern implementation errors—will be addressed in the next chapter, where we present the details of the MonGen tool. MonGen automates the creation of runtime pattern contract monitors, given a set of PCL contracts and subcontracts for a particular system. We will often mention MonGen in this chapter when discussing the design details of the PCL language, since they have been influenced by monitoring considerations and the development of the tool.

Several authors have considered how to formalize design patterns, and we will discuss a number of these proposed approaches in Sec. 5.6. At this point, however, we will note that our work is unique in two important ways. First, it is the first to propose a specification language that precisely captures the implementation requirements and behavioral guarantees associated with a range of patterns, while simultaneously accommodating the variation that occurs across applications of the same pattern. Second, our work is the first to consider automating the generation of monitoring code for detecting pattern implementation errors based on the relevant pattern specifications.
5.2 Flexibility and Specificity in Pattern Contracts

In our discussions of both specification and monitoring, we will occasionally refer to the Observer pattern to make some of the ideas presented more concrete; we will formally present its PCL contract later in Sec. 5.4.1. As previously mentioned in the introductory chapter, the intent of the Observer pattern is to maintain consistency between a subject object and a set of observer objects which are interested in its state. The informal description of the pattern makes it clear that an object must invoke an Attach() method to become an observer of a particular subject, and Detach() when it is no longer interested in the subject. The description also states that a subject invokes its Notify() method whenever a change occurs that could make its observers’ states inconsistent with its own, and that Notify() itself invokes Update() on each of the attached observers. Such a description, albeit useful for gaining a general understanding as to what the pattern should do, does not provide a formal way to express how a specific application of the pattern may meet these constraints.

One might argue that pattern descriptions must remain ambiguous in order to allow for flexibility in their application, and that formalization efforts run the risk of compromising this flexibility [68]. If, for example, we were to adopt a single definition for the notion of consistency, the pattern would not be applicable in systems that have a different notion of this concept. PCL, however, is designed to preserve essential flexibility while simultaneously eliminating accidental ambiguity. Furthermore, this formalization often helps identify additional dimensions of flexibility that

19We use names starting with uppercase letters, such as Subject, for roles, and corresponding lowercase names, such as subject, to refer to the objects that play these roles. In some cases the name of a pattern is also used for one of the constituent roles, as in the case of the Observer role of the Observer pattern. In such cases, the context will make clear which is intended.
are not mentioned in the informal descriptions, which we will see later in an example. This flexibility is achieved by separating what is general to the pattern, and what is specific to the particular application into contracts and subcontracts. The pattern contracts are parameterized using the following, which are in turn instantiated in the subcontracts for the particular pattern’s application:

- **Role maps.** The roles defined in the pattern contract are described in terms of fields and methods, which may not exactly match up in name or structure to their counterparts in the class objects which play these roles in the application. The role maps in the subcontract define how the elements of the class objects correspond to those in the roles in the contract.

- **Auxiliary concepts.** An auxiliary concept is a relation involving one or more states of the objects participating in the pattern, which corresponds to a point of variation explicitly expressed—or in some cases implicitly allowed—by the informal pattern description.

For example, the contract for the Observer pattern defines the Subject and Observer roles. Suppose, now, we have a Hospital application that contains Patients and Nurses that play these respective roles. The subcontract for this Hospital application would define mappings from a Patient’s concrete state and methods, to those for Subjects as found in the contract. Similar mappings would be found between Nurses and the Observer role. The Observer contract will also declare two auxiliary concepts: Modified() and Consistent(). The first concept will capture whether the state of the subject has been modified in a way that could result in inconsistency with an attached observer, and the second will capture whether the state of the subject is consistent
with the state of a particular observer. In the subcontract, a concrete definition for \textit{Modified()} would be provided that tells if a Patient's state may have changed enough to result in an inconsistency, and a concrete definition for \textit{Consistent()} would tell us if a Patient's state is consistent with an attached Nurse's state. We will elaborate on this particular example in Sec. 5.4.2, after we first go through the details of the PCL language.

5.3 The Pattern Contract Language

5.3.1 Pattern Contract Grammar

A design pattern contract consists of a number of roles, which can contain both state and methods. One set of requirements that the pattern can impose is how these roles interact in specific ways. In the Observer pattern, one such requirement, is that when the state of the subject changes and its new state might be inconsistent with that of its observers, it should invoke the \textit{Update()} operation on each observer. In the PCL formalism, these requirements will be part of the role contract corresponding to the Subject role. In general, the contract for a given pattern will include a role contract corresponding to each role of the pattern, in addition to the state information for each role. The pattern contract will also specify a \textit{pattern invariant}, an assertion involving the states of all the roles. The invariant will be satisfied whenever control is outside all of the methods of all the roles. The invariant is, essentially, the “defined properties” [18] that the correct use of this pattern ensures for the system. The role contracts, as well as the invariant, will be expressed using auxiliary concepts. As noted earlier, however, the definitions of the concepts will not be part of the pattern contract, but will instead be provided in the subcontracts.
Although the definitions of the concepts in the subcontracts may be tailored in
whatever way is necessary to meet the needs of a particular application, it often turns
out that if a pattern contract involves more than one concept, then these concepts’
definitions in any application must satisfy certain *constraints*. Otherwise, the intent
of the pattern will be violated even if the system meets all other requirements of the
pattern. We will see this in our contract for the *Observer* pattern. Thus an important
part of the pattern contract will be the constraints that the various auxiliary concepts
used in the contract must satisfy.

A pattern contract must also specify how new instances of the pattern will be
created, which is done via the *instantiation* clause. We call any group of objects
interacting according to a given pattern an *instance* of the pattern. Each pattern
instance will have a *lead object* that acts as the unique identifier for that particular
instance. A pattern contract will designate one of its roles as the *lead role*, from which
the lead object comes. For example, in a particular system built using the *Observer*
pattern, at any given point during execution, we may have several groups of objects,
with each group consisting of one object playing the *Subject* role, and the other
objects playing the *Observer* role that are “observing” the first object. Each such
group is an instance of the *Observer* pattern. In the contract for the *Observer*
pattern, the instantiation clause specifies that a new pattern instance is created when a new
object playing the *Subject* role is constructed—this object will be the lead object for
this instance. The *Subject* role is designated as the lead role for the pattern.

Note, however, that the group of objects in a given instance is not static, and
so the contract must specify how an object can join a particular pattern instance.
This is accomplished using an *enrollment* clause for each role of the pattern. Again,
using the Observer pattern, a new object may wish to start observing a Subject object (which may already have several other objects observing it). This object may do so by invoking the Attach() method on the particular subject. This will be specified, in our contract for the pattern, as the enrollment clause for this role. Because there may be multiple instances of the Observer pattern present in the system, it is important that we say specifically which instance the object is enrolling in. This is done by referring to the lead object for that particular instance in the enrollment clause.

The first part of the grammar\textsuperscript{20} for pattern contracts appears in Fig. 5.1. A pattern contract consists of the name of the pattern (⟨pid⟩), the auxiliary concepts

\textsuperscript{20}In our presentation, when we use plural names for nonterminals, it is implied that there is a rule where the plural nonterminal is defined as a sequence of nonterminals using the singular name.
needed, the state information for each role, the constraints that must be satisfied by the definitions of the auxiliary concepts, the instantiation clause, an optional deinstantiation clause, the pattern invariant, and the role contract for each role.

Corresponding to each auxiliary concept, we specify its name \(\langle cid\rangle\), and the list of role names that the concept is concerned with. We considered the possibility of allowing default definitions for some or all the concepts in the pattern contract, but decided against it because of the risk that a needed definition might be omitted from a subcontract by mistake. Next, we specify the state of each of the roles using the rule for \(\langle roleState\rangle\), where the lead keyword should be used if \(\langle rid\rangle\) is the lead role of the pattern. We use a type name \(\langle typeName\rangle\), variable name \(\langle sid\rangle\) pair for each of the state fields of a role. In PCL contracts, a type can be a class, an interface, or a role.

The rule \(\langle adjState\rangle\) defines the adjunct state for the given role \(\langle rid\rangle\). The adjunct state for a role object is state associated with the object that plays a part in specification, yet is usually not part of the object’s internal state. In this way, adjunct state variables are very similar to the ghost variables of JML [54]. Because this state information generally cannot be extracted from the object, it is not (directly) defined in the role maps of subcontracts. Instead, it is defined as part of the contract itself, where the adjunct variables are set when specified methods complete execution; this is done using the set adjunct clauses in the \(\langle reqPresEns\rangle\) rule. It should be mentioned that adjunct state variables are not meant as a replacement for traces, which are already included in the PCL formalism, and are maintained by the monitors generated by the MonGen tool. The intended use of adjunct state variables is to define local
properties associated with individual role objects, and not global properties, as traces do.

Next, we have the rule for \( \langle \text{simpleRole} \rangle \), which allows us to define simple roles. A simple role does not have any role state, adjunct state, or methods associated with it, and the class objects that will play the role will not join a pattern instance. Simple roles act as a type that is used for parameters and return values in the specifications, and are mapped to actual application types in the subcontracts. After \( \langle \text{simpleRole} \rangle \), we have the set of constraints that must be satisfied by the auxiliary concepts, as they are defined in any subcontract of this contract. After that, we have the instantiation clause, which specifies the particular role method that must be invoked to create a new instance of the pattern. Here, the signature of the method is given in \( \langle \text{roleMethSig} \rangle \)—note that the \( \langle \text{roleMethSig} \rangle \) rule includes the role in which the method resides (\( \langle \text{rid} \rangle \)). In many cases, a new instance of the pattern is created as soon as an object that will play the particular role is constructed, rather than at a later point when a method is invoked on that object. In such a case, \( \langle \text{methId} \rangle \) will be specified as the ‘method’ new, representing a call to the appropriate constructor. The instantiation clause also provides for the specification of pre- and postconditions, as well as a list of preserved variables (\( \langle \text{reqPresEns} \rangle \)). Only one instantiation clause appears in a pattern contract; multiple clauses could have been allowed, but in the patterns we have studied this generality was not needed. The deinstantiation clause, in a similar fashion, specifies when a particular pattern instance should no longer be considered a pattern instance.

The next two rules are relatively straightforward. The invariant clause specifies the pattern’s invariant, as we have described. The next rule is used for defining the
\[
\langle \text{roleContract} \rangle \ ::= \ \text{lead}? \ \text{role} \ \langle \text{id} \rangle \ \text{contract} \{ \\
\langle \text{enrollment} \rangle)? \ \langle \text{disenrollment} \rangle)? \ \langle \text{namedMethSpecs} \rangle \ \langle \text{othersSpec} \rangle \}
\]

\[
\langle \text{enrollment} \rangle \ ::= \ \text{enroll} : \ \langle \text{roleMethSig} \rangle \{ \ \text{lead} : \langle \text{lid} \rangle ; \ \text{enrollee} : \langle \text{eid} \rangle ; \ \langle \text{reqPresEns} \rangle \}
\]

\[
\langle \text{disenrollment} \rangle \ ::= \ \text{disenroll} : \ \langle \text{roleMethSig} \rangle \{ \ \text{lead} : \langle \text{lid} \rangle ; \ \text{enrollee} : \langle \text{eid} \rangle ; \ \langle \text{reqPresEns} \rangle \}
\]

\[
\langle \text{namedMethodSpec} \rangle \ ::= \ \langle \text{retType} \rangle)? \ \langle \text{methId} \rangle ( \ \langle \text{args} \rangle ) : \ \langle \text{reqPresEns} \rangle
\]

\[
\langle \text{othersSpec} \rangle \ ::= \ \text{others} : \ \langle \text{reqPresEns} \rangle
\]

Figure 5.2: PCL contract grammar (part 2)

signatures for role methods. Note that \( \langle \text{retType} \rangle \) is optional—if it is omitted, the method is treated as if it does not return a value.

The last rule lists the requires, preserves, ensures, and set adjunct clauses, where only the ensures clause is non-optional. The semantics of the preserves clause is different than the one-deep equality used in Chapters 2 and 3. From application to application, the structure of the internal state of the objects playing a particular role may be quite different, and so the resulting requirement on concrete state may not be uniform for different systems if we used the same semantics as before. Furthermore, this would put restrictions on the objects’ concrete state in the pattern contract, which is supposed to deal with role states, not concrete states. Instead, we use simple reference equality for non-container typed variables, and reference equality among the items of container objects to determine if they are preserved.

Let us now consider the role contract corresponding to the role \( \langle \text{id} \rangle \) in the second part of the contract grammar\(^{21}\), shown in Fig. 5.2. Here, if \( \langle \text{id} \rangle \) is designated as the lead, then the enrollment clause is optional for this role. This is because lead

\(^{21}\)Note that the role state is not specified as part of the role contract, but given much earlier in the pattern contract. This allows MonGen to parse the contracts and process the monitoring aspects in a single pass.
role objects become part of a pattern instance when a pattern is instantiated, which is specified in the instantiation clause for the pattern. If additional objects of the lead role can take part in a pattern instance as a non-lead object, they will do so by calling the particular method (or constructor) specified in the enrollment clause for this role. Or, if additional objects of the lead role are not allowed to enroll in the same instance—as is the case with the Subject role of Observer in which each instance of the pattern has only one subject enrolled—then the enrollment clause will be missing from the role contract for this role. In any case, the enrollment clause resembles the instantiation clause, in that a call to a particular role’s method (or constructor) is designated, as well as the lead object of the instance in question. The enrollment clause must, of course, also identify the enrolling object.

The disenrollment clause serves a similar purpose as the disinstantiation clause, where it specifies when a particular role object no longer plays a part in a given pattern instance. If the role object to be disenrolled is the lead object of the given pattern instance, this should not be specified here—that is the responsibility of the disinstantiation clause.

Following the enrollment clause, we have specifications of the named methods and the ‘other’ methods. The named methods are the ones that this role must have in order to play its part in the pattern. For example, the Observer role of the Observer pattern must provide an Update() method; that is one of the named methods of this role. Suppose, in a specialization of a given pattern P, the class C plays the role R. The methods of C that correspond to the named methods of R will be required to satisfy the corresponding specifications in R’s role contract. But, in general, C will provide additional methods; these are its other methods. If these methods are
not suitably designed, then the intent of the pattern may be compromised. For example, the role state for Subject will specify that this state must include a variable whose value will be a set of references to the objects enrolled to observe the subject. Now if a class playing the Subject role were to include a method that changed or destroyed the information in this variable, the system would clearly fail, even if all the methods explicitly listed in the Subject role were implemented correctly. The others specification imposes conditions on the methods that are not explicitly listed in the contract to prevent such problems.

The boolean expressions in the contracts, ⟨consExpr⟩, ⟨invExpr⟩, ⟨reqExpr⟩, and ⟨ensExpr⟩, and the general expression ⟨valExpr⟩, are written in the same expression language for assertions described in Chapters 2 and 3. There are slight differences in the semantics, though, to better suit pattern specification and monitoring. In these expressions, we can refer to the state variables of a role object in exactly the same way we refer to the fields of a class object, using the familiar dot-notation, and as before, we can refer to ‘old’ values using the ‘#’ operator. These role state variables are used to help model the role object, and are not necessarily fields found in the actual class of the object. As we have mentioned, role state values are projected from the concrete object state using the role maps defined in the subcontracts. In this scheme, #-ed role state variables represent the value projected from the previous state of the object.

In PCL, we can also refer to the trace variable τ in postconditions. However, here, τ captures all of the calls made by a method on role objects, not just those calls made to hooks. Also, the methods (and their arguments) referred to in traces here are the role methods listed in the contract, and not application-specific methods. In
our specification language, we can refer to sub-traces of \( \tau \) that only contain calls to a particular method using a given target object. For instance, \( \tau.\text{obj}.\text{meth} \) gives us the sub-trace containing only those calls to method \( \text{meth}() \) on an object \( \text{obj} \).

Special notation is provided for explicitly referring to the role objects that play a part in the pattern instance, using the keyword \textit{instance} to refer to the instance. With the dot notation, we can get the set of all objects in the instance playing a particular role. For example, \texttt{instance.Observer} gives us the set of objects playing the \texttt{Observer} role in the instance. We can also refer to the lead object of the pattern instance with \texttt{instance.lead}. This notation is primarily used in the \texttt{invExpr}s, because pattern invariants specifically impose conditions on pattern instances; pre- and postconditions are mostly concerned with the objects involved with a method invocation, and not so much with the other objects in the instance. The constraint expression, \( \langle \text{consExpr} \rangle \), conveys properties that the auxiliary concepts should adhere to, and generally involves quantifications that span over all possible object states of a role or class. For this reason, constraints are not checked at runtime by the monitoring tool.

### 5.3.2 Subcontracts

The grammar for subcontracts appears in Fig. 5.3. A subcontract lists the name of the specialization, the name of the pattern it specializes, provides a set of role maps (one corresponding to each application class whose objects will play a role of the pattern), and a set of definitions for the auxiliary concepts of the pattern. A non-simple role map specifies how a particular class \( \langle \text{cid} \rangle \) plays the part of a particular role \( \langle \text{rid} \rangle \) of the pattern. The \textit{state map} and \textit{interface map} together tell us exactly how we can think of objects that are instances of this class as instances of this role.
\(\langle \text{subContract} \rangle ::= \text{subpattern} \langle \text{sid} \rangle \text{specializes} \langle \text{pid} \rangle \{\ \langle \text{simpleRoleMaps} \rangle? \ \langle \text{roleMaps} \rangle \ \langle \text{auxConceptDefs} \rangle \}\)

\(\langle \text{simpleRoleMap} \rangle ::= \text{rolemap} \langle \text{cid} \rangle \text{as} \langle \text{srid} \rangle ;\)

\(\langle \text{roleMap} \rangle ::= \text{rolemap} \langle \text{cid} \rangle \text{as} \langle \text{rid} \rangle \{\ \langle \text{stateMap} \rangle \ \langle \text{interfaceMap} \rangle \}\)

\(\langle \text{stateMap} \rangle ::= \text{state:} \{\ \langle \text{varMaps} \rangle \}\)

\(\langle \text{varMap} \rangle ::= \langle \text{rvid} \rangle = \{\ \langle \text{returnCode} \rangle \}\)

\(\langle \text{interfaceMap} \rangle ::= \text{methods:} \{\ \langle \text{methodMaps} \rangle \}\)

\(\langle \text{methodMap} \rangle ::= \langle \text{rmid} \rangle(\langle \text{rmargs} \rangle) : \langle \text{cmid} \rangle(\langle \text{cmargs} \rangle) ;\)

\(\langle \text{auxConceptDef} \rangle ::= \text{auxiliary concept} \langle \text{auxid} \rangle(\langle \text{cargs} \rangle) \{\ \langle \text{returnCode} \rangle \}\)

Figure 5.3: PCL subcontract grammar

The state map is a set of variable maps, one corresponding to each variable listed in the role contract for this role in the pattern contract. A variable map lists the name of the particular variable defined in the role (\(\langle \text{rvid} \rangle\)), and the Java code that takes the current state of the \(\langle \text{cid} \rangle\) object and returns the value of this particular role variable when the \(\langle \text{cid} \rangle\) object is viewed as an \(\langle \text{rid} \rangle\) object. Thus the variables listed in the role contract for \(\langle \text{rid} \rangle\) in the pattern contract do not have to be part of \(\langle \text{cid} \rangle\). Instead, given any state of the \(\langle \text{cid} \rangle\) object, the code listed in the \(\langle \text{varMap} \rangle\) corresponding to any variable of the \(\langle \text{rid} \rangle\) role will return the value of that variable when we view the \(\langle \text{cid} \rangle\) object in this role. In simple cases, each role variable will correspond to a variable in the class, and the code will simply return the value of that variable. However, PCL allows the writer of the subcontract to define more complex maps using code, as we will see in the examples in Sec. 5.4.

We should mention that it is quite possible that an object playing a particular role can participate in multiple instances of the same pattern. For instance, we could
have a Nurse object that observes multiple Patient objects. Here, the Nurse is playing the role of Observer for several pattern instances, with each pattern instance having the Patient (the Subject) object as the lead object. Details of this example will be discussed in full in Sec. 5.4.2; but for now, we will just state that PCL can handle this possibility by giving us a means to essentially project the role state for a given application object relative to a single pattern instance. This is accomplished by referring to the lead object of the particular pattern instance, using the lead keyword in the ⟨varMap⟩ code. We will return to this when going through the examples.

The method maps in the interface map define the mappings between the methods of the class and the methods of the role as listed in the pattern contract. For each method map, ⟨rmid⟩ is the name of the role method with ⟨rmargs⟩ as its (typed) parameters, and ⟨cmid⟩ is the name of the corresponding class method with ⟨cmargs⟩ as its (typed) parameters. To match up a role method and class method parameter, the same parameter name should be used for both.

In any given specialization of the pattern, more than one class may play a given role. For example, in the Hospital application that we will discuss in detail in Sec. 5.4.2, the Patient class will play the Subject role, while the Observer role will be played by two classes, Nurse and Doctor (since both nurse objects as well as doctor objects “observe” patient objects). Such specializations of the pattern are allowed by PCL.

Following the role maps, we have the auxiliary concept definitions. For each auxiliary concept ⟨auxid⟩, and each possible combination of classes that play the various roles that appear as its parameters, we must provide a definition. This is done by providing suitable code that performs appropriate comparisons of the objects of
the particular classes mapped to each role, and returns a true or false value to indicate if the relation represented by the concept is satisfied or not. An alternative approach would have been to require the concept definitions to be expressed as mathematical expressions, such as in our pre- and postconditions. However, we decided to use code here, as well as in the \(\text{varMap}\)s, on practical grounds. First, using code here simplifies the processing task of MonGen. Second, subcontracts have to be tailored to their specific applications, and so will most likely have to be written by general practitioners who may not be as comfortable writing formal specifications as they are writing code. Pattern contracts, on the other hand, only have to be written once, and can be applied to any application that implements the pattern. It only requires a few people comfortable with formal specifications to write contracts, which can be made freely available to the public.

5.4 Pattern Contract and Subcontract Examples

5.4.1 Observer Pattern Contract

Here, we present the PCL contract for the Observer pattern. In the next section, we will consider a simple application, Hospital, and develop the subcontract corresponding to its specialization of Observer. The first part of the specification of the Observer pattern appears in Fig. 5.4. The contract uses two auxiliary concepts, Consistent and Modified. Consistent represents the notion that a given subject state is consistent with a given observer state. Modified represents what it means for a given subject state to be sufficiently different from another state of the subject. By sufficiently different, we mean different enough in that it would require the notification of the observers if the subject went from the one state to the other. The next two
pattern Observer contract {
    concept Consistent(Subject, Observer);
    concept Modified(Subject, Subject);

    lead role Subject state { Set _observers; }
    role Observer state { Subject _subject; }

    constraint:
        Modified : irreflexive;
        (\forall Subject s1, Subject s2, Observer o1 ::
            (\neg Modified(s1, s2) \land Consistent(s1, o1)) \Rightarrow Consistent(s2, o1))

    instantiation: Subject.new() {
        lead: result;
        ensures: result._observers = \emptyset }

    invariant:
        instance.Subject = { instance.lead }
        \land (\forall Observer ob \in instance.Observer ::
            ob._subject == instance.lead \land Consistent(ob._subject, ob))

    ...role contracts for Subject and Observer, see Figs. 5.5 and 5.6 ...
}

Figure 5.4: Observer pattern contract
lines give us the role state for objects playing the Subject and Observer roles. The role state for Subject consists of a single variable, _observers, whose value will be the set of references to all the objects currently attached to observe this subject. Likewise, an observer object should have a reference to what it is observing—which is given by the _subject variable for the Observer role.

Let us now consider the constraint clause. First, it states that the Modified() concept is an irreflexive relation. The second part states that if a subject changes state from s1 to s2, but the change is not sufficiently different in the aforementioned sense, and s1 is consistent with an Observer o1, then s2 should also be consistent with o2. To illustrate why the constraint clause is important to the pattern, suppose we have definitions for Consistent() and Modified() in a subcontract where the constraint does not hold. Here, we could have a subject s1 be consistent (according to the given definition of Consistent()) with one of its observers o1, where its state could change from s1 to s2 where Consistent(s2, o1 is false, but o1 would not be notified because Modified(s1, s2 is also false. The problem here arises not because of the failure of the subject or the observers to interact in the manner intended by the pattern, but rather having mutually incompatible notions of consistency between the state of a subject and an observer, and what it means for the state of the subject to be materially modified. The constraints in our pattern contracts serve as explicit restrictions on the auxiliary concepts to prevent such incompatibilities. However, as was previously mentioned, constraints cannot be effectively checked at runtime because they state general properties on concepts (which usually contain unbounded universal quantification in their specification), and not specific behaviors of the system. For this reason, MonGen does not produce checking code for them.
Let us turn to the instantiation clause. It specifies that a new instance of the pattern will be created when an instance of the Subject role is constructed, and that this new instance will have as its lead the newly constructed object. The condition specified states that following this construction, the value of the _observers field of the role state of the newly constructed subject must be empty. This condition represents the fact that when an object that will play the subject role is created, no observers have enrolled to observe it.

Next we have the pattern invariant. The first clause states that the set of objects playing the Subject role in the pattern instance is the singleton set containing the lead object. The next clause states that for every Observer ob taking part in the pattern instance, ob’s _subject is the lead object, and that ob is consistent with it.

The contract for the Subject role appears in Fig. 5.5. There is no enrollment clause here because the only way for a Subject object to take part in a pattern instance is by being the lead object in an instantiation. The role has three named methods, Attach(), Detach(), and Notify(). Attach() is invoked when an object ob wishes to become an observer of this subject. As specified in the requires clause, this method may only be called if ob is not already enrolled in the pattern instance—i.e., not one of the subject’s observers. The first conjunct of the ensures clause specifies that the enrolling observer is added to the _observers set. The next conjunct requires that the state of the subject not be Modified, which may seem strange since we just stated that the _observers field must be modified. The point is that this change has to do with how the subject keeps track of which objects are observing it, and is not the kind of change the observers would be interested in. Hence, this kind of change should not be significant as far as the Modified() concept is concerned.
lead role Subject contract 

```java
void Attach(Observer ob):
    requires: ob \notin _observers
    ensures:
        (_observers = #_observers \cup \{ob\}) \land \neg Modified(#this, this)
        \land Consistent(this, ob)

void Detach(Observer ob):
    requires: ob \in _observers
    ensures:
        \neg Modified(#this, this) \land (_observers = #_observers \setminus \{ob\})

void Notify( ):
    preserves: _observers
    ensures:
        \neg Modified(#this, this) \land |\tau| = _observers
        \land (\forall ob \in _observers : |\tau.ob.Update| = 1)

others:
    preserves: _observers
    ensures:
        (\neg Modified(#this, this) \land |\tau| = 0)
        \lor (|\tau| = 1 \land |\tau.this.Notify| = 1)
```

Figure 5.5: Subject role contract
Indeed, definitions of auxiliary concepts commonly ignore the ‘pattern-portion’ of the states of the participating objects. The condition captured by the last conjunct 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 of the given subject properly updated. The Attach() method adds a new observer, and therefore, needs to ensure that it is consistent with its subject\(^{22}\).

For the Detach() method, we require that the detaching observer be attached to the subject at the time of the call. When the method terminates, the detaching observer must be removed from _observers, and the subject state must not be modified—according to the concept Modified()—from what it was at the start of the method.

The final named method is Notify(), whose purpose is to update each observer. This method, as we will see in the others specification, is required to be called by any method that modifies the state of the subject. The preserves clause requires that the references contained in _observers not be changed. The ensures clause first states that Notify() not modify the state of the subject, and its last two conjuncts state that the Update() method must be invoked on each attached observer.

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

\(^{22}\)In some of our previous versions of the Observer contract, the postcondition here stipulated that Attach() must call the Update() method in an effort to achieve consistency. However, we have found reasonable examples that accomplish this without invoking Update(), and so it is not part of the contract here.
role Observer contract {
    enrollment: Subject.Attach(ob) {
        lead: target;
        enrollee: ob;
        ensures: true
    }
    disenrollment: Subject.Detach(ob) {
        lead: target;
        enrollee: ob;
        ensures: true
    }
    void Update():
        preserves: _subject
        ensures: Consistent(_subject, this)
    others:
        preserves: _subject
        ensures: Consistent(_subject, #this) ⇒ Consistent(_subject, this)
}

Figure 5.6: Observer role contract

Now, let us turn to the role contract for the Observer role, shown in Fig. 5.6. The enrollment clause states that, for an object to enroll in this role, the Attach() method must be invoked on the appropriate subject, which will be the lead object of the pattern instance, and the enrolling observer must be passed as the argument. The disenrollment clause is similar.

There is only one named method, Update() in this role. Its specification first states in its preserves clause that the reference to the subject being observed is not lost. Its postcondition simply states that the method should make the state of the observer consistent with that of the subject.

Last we have the others specification. It requires _subject remain unchanged, since otherwise the observer would lose its reference to the subject it is observing. The
ensures clause requires that if the state of the observer at the start of the method was consistent with its subject, then the state of the observer at the end of the method also be consistent with subject. This captures a critical aspect of the Observer role’s behavior. It allows an others method of the role to modify the state of the observer as long as the modification does not affect the consistency of this state with that of the subject state. Standard descriptions of the pattern suggest that the state of an observer should not change except when the Update() method is invoked, which is unnecessarily restrictive. Suppose, for example, the observer in question were a type of display object that displays, in the form of a pie-chart, information it has about the state of the subject. Suppose also that this observer provides the option allowing it to be iconified and de-iconified. In going from being de-iconified, with the pie-chart information about the subject being displayed, to being iconified, clearly no information about the subject is lost. 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. And it will be in our formalism, if we define the predicate Consistent() appropriately (in the subcontract). By contrast, standard descriptions of the pattern would seem not to allow such changes in the object playing the Observer role. Once we recognize this, we can, of course, rewrite the informal description to permit such changes. But the point remains that it is the process of formalization that allowed us to identify this dimension of flexibility that is missing in standard descriptions.
5.4.2 Hospital Application and Subcontract

We now present a brief example that uses the Observer pattern, the Hospital system. There are three main classes in the system, Patient, Doctor, and Nurse. We first start with the Patient class, shown in Fig. 5.7, whose objects play the role of Subject, and can be observed by zero or more Nurse class objects (stored in the set nurses), and zero or one Doctor class objects doc. The patient’s internal state includes its body temperature (temp), heart rate (htRate), and medication level (medLvl). The class contains ‘getter’ and ‘mutator’ methods for these fields. A nurse is assigned to a patient by invoking the addNurse() operation on the patient, passing the nurse as an argument; similarly a doctor is assigned by invoking addDoctor(). The pageAll() method is used to update all of the nurses and the doctor observing the patient. Note that tempChange and htRateChange() both invoke pageAll(), whereas medLvlChange does not.

The Nurse and Doctor classes are both shown in Fig. 5.8. A nurse can only monitor a single patient and its temperature, whereas a doctor can observe multiple patients. The Doctor class contains a Java HashMap that maps its patients to their condition. A ‘true’ value here means that their condition is good with a temperature below 100 and heart rate below 85; otherwise, the value is ‘false’, meaning a bad condition. The Nurse class has a method watch() which associates the patient argument with the nurse and stores its temperature in the patTemp variable. In the Doctor class, the oversee() method plays a similar function, but must do so by updating its pat2cond hash map. Both classes also have an updating method (updTemp() and updCond()), which is to be called by pageAll() in the Patient class, which itself is called when the state of the patient undergoes a meaningful change.

137
class Patient {
    protected int temp, htRate, medLvl;
    protected HashSet nurses;
    protected Doctor doc;

    Patient(int t, int h, int m) {
        temp = t; htRate = h; medLvl = m;
        doc = null; nurses = new HashSet();
    }

    protected int getTemp() { return temp; }
    ... other standard getters: getHtRate(), getMedLvl() ...

    public void addNurse(Nurse n) {
        nurses.add(n); n.watch(this);
    }

    public void addDoc(Doctor d) {
        doc = d; d.oversee(this);
    }

    public void tempChange(int delta) {
        temp += delta; pageAll();
    }

    public void htRateChange(int delta) {
        htRate += delta; pageAll();
    }

    public void medLvlChange(int delta) {
        medLvl += delta;
    }

    protected void pageAll() {
        Iterator it = nurses.iterator();
        while (it.hasNext()) {
            ((Nurse)(it.next())).updTemp();
        }
        if (doc != null) {
            doc.updCond(this);
        }
    }
}

Figure 5.7: Patient class code
class Nurse {
    protected Patient pat;
    protected int patTemp;
    Nurse() { pat = null; patTemp = 0; }
    public void watch(Patient p) {
        pat = p, patTemp = p.getTemp();
    }
    public void updTemp() { patTemp = p.getTemp(); }
}

class Doctor {
    protected HashMap pat2cond;
    Doctor() { pat2cond = new HashMap(); }
    public void oversee(Patient p) {
        Boolean cond = new Boolean(p.getTemp() < 100 && p.getHtRate() < 85);
        pat2cond.put(p, cond);
    }
    public void updCond(Patient p) {
        if (pat2cond.keySet().contains(p))
            oversee(p);
    }
}

Figure 5.8: Nurse and Doctor class code
The subcontract for the Hospital system is shown in Fig. 5.9. In this system, Patient plays the Subject role, and Doctor and Nurse play the Observer role. First, let us consider the state maps of the role maps. For the Patient role, we must essentially collect all of the nurses and the doctor that are observing it into a single set, and return it in the definition for observers. For an object playing the Nurse role, only a single patient can be observed, so the state map for subject simply returns the reference to the pat field object. The state map for Doctor is more interesting. Because a doctor can observe several patients, and hence, participate in several instances of the Observer pattern, it becomes necessary to determine which subject is being referred to relative to a given pattern instance. This is where the special parameter lead is used to represent the pattern instance. Because lead would also be the patient that the doctor would be observing in this case, we simply return lead. The method maps are relatively straightforward for all three classes.

Now, for the auxiliary concept definitions. The Modified() concept is defined so that it represents a change in the value of temp or htRate—because a change in the medLvl would not warrant an update of any of the observers of the patient, this is not included in the definition. A nurse state is defined to be consistent with the state of the patient if the temperature information in the nurse’s patTemp matches its temp value. A doctor’s state is defined to be consistent with the state of a given patient if the condition stored in its hash map for this patient correlates to its temperature and heart rate information.

Given this subcontract, we can now say definite things about what is required of the Hospital system with respect to the Observer contract. First, note that the definitions for Consistent() and Modified() given in the subcontract should adhere to the
subpattern Hospital specializes Observer {
    rolemap Patient as Subject {
        state: { _observers = { Set s = new HashSet(this.nurses);
                            if(this.doc != null) s.add(this.doc);
                            return s; } }
        methods: { Attach(Observer ob) : addNurse(Nurse ob), addDoc(Doctor ob);
                    Notify() : pageAll(); }
    }

    rolemap Nurse as Observer {
        state: { _subject = { return pat; } }
        methods: { Update() : updTemp(); }
    }

    rolemap Doctor as Observer {
        state: { _subject = { return lead; } }
        methods: { Update() : updCond(Patient p); }
    }

    auxiliary concept Modified (Patient s1, Patient s2) {
        return s1.temp != s2.temp || s1.htRate != s2.htRate;
    }

    auxiliary concept Consistent (Patient s, Nurse o) {
        return o.patTemp == s.temp;
    }

    auxiliary concept Consistent (Patient s, Doctor o) {
        Boolean cond = ((Boolean)(o.pat2cond.get(s)));
        return cond.booleanValue() == (s.temp < 100 && s.htRate < 85);
    }
}

Figure 5.9: Hospital subcontract
constraint clause specified in the contract, which they do. The invariant of Observer stipulates that before and after an outside method call is made on a participant in the Hospital system, each patient should be consistent (as defined in the subcontract) with its attached nurses and doc. As an example of a named method, when pageAll() is invoked on a given patient, we apply the specification for Notify() found in the Subject role contract, since the subcontract maps pageAll() to Notify(). From this, we can deduce that its set of nurses and doc should not be changed by the method (the preserves clause), its temp and htRate should also not be changed (the ¬Modified() condition in the postcondition), the updTemp() method should be invoked on each of its nurses exactly once, and the updCond() method should be invoked on its doc once (the trace conditions in the postcondition). The class methods that are not mapped to a role method in the subcontract, but whose classes play a role in the pattern, are categorized as other methods. Here, these include the setter and mutator methods of the Patient class, as well as the methods Nurse.watch() and Doctor.oversee(); these should adhere to the others specifications given in their respective role contracts.

Now suppose that the designers of this system modify it, such that doctors now determine their patients condition based on heart rate and medication level, instead of heart rate and temperature. The notion of consistency will naturally be revised to require that in each doctor’s pat2cond field, each of their patients’ recorded condition matches their condition based on the current value stored in the patient’s medLvl and temp fields. This would be reflected in a new definition for the Consistent() auxiliary concept. The designers would also presumably revise Doctor.oversee() (which is used by Doctor.updCond() to determine condition) to record the patient’s condition according to the new guidelines. However, this is not sufficient. If changes are mode
to a patient’s medLvl via medLvlChange(), the necessary calls to pageAll() will not be triggered. As a result, calls to medLvlChange() may leave doctors with inconsistent views of their patients.

With respect to the pattern, the only change dictated by the new requirements would seem to be a redefinition of what it means for the state of a doctor to be consistent with the state of a patient, along with the corresponding changes in Doctor.oversee(). However, as we have seen, this is inaccurate. The change in the notion of consistency requires a corresponding change in the notion of significant change in a patient’s state. In the Hospital system, a change in medLvl is now significant, and should therefore trigger a call to pageAll(). These critical relationships between the concepts used in describing a pattern are captured in the constraint clauses of the pattern contract. In this case, we need to revise our definition of Modified() to fit this new definition of Consistent(), to take into consideration changes in medLvl.

5.4.3 Memento

The intent of the Memento pattern [33] is to allow an originator object to save its state in the form of an external memento object that can be later used to restore the originator’s state. The memento objects are created using the CreateMemento() method of the Originator role, and when the originator is to be restored to a previous state, the memento which stores the desired state is passed as the argument to the originator’s SetMemento() method.

The internal state of the memento objects should contain sufficient information about the original state of the originator (original relative to when the memento was created) so that SetMemento() can restore the originator’s state; the memento is not
required to store the originator’s full state. From this requirement, it is reasonable to allow the memento’s internal state to change—including the state components that relate to the saved originator state—as long as the change is not enough to prevent the original originator’s state to be restored from the memento. Many of the auxiliary concepts were introduced in the PCL contract (shown in Fig. 5.10) to allow for this flexibility. The intent of the auxiliary concept MemCopy() is to express that the given memento contains in its state a faithful representation of the given originator’s state. The idea behind SameMem() is closely related, as seen in the first quantified expression in the constraint. SameMem() is used to express that if a memento’s internal state changes, the change is not sufficient enough to prevent it from restoring the originator to its original state. The StateInfo() concept is only used to help specify the GetState() method of the Memento role, which we will described later. Reset() captures the idea that if an originator is restored using a given memento, the originator’s state listed as the third argument would be a valid result of the restore. The final concept, Restored(), says that the second originator argument is a valid restored version of the first originator argument.

Next, we list the role state clauses, which do not contain anything for either role. At first, this may seem surprising, but consider what information each of the role objects should contain. First, for Originator, we might think that it should contain the set of mementos created from it. However, there is no requirement that this information be stored by the originator objects in some form or another, and so it is not part of the role state. (We can access this information, though, via the instantiation keyword, which we will see later). Another thing that appears to be missing is in the role state for Memento, where there is no mention of the internally saved state of
pattern Memento contract {
  concept MemCopy(Originator, Memento);
  concept SameMem(Memento, Memento);
  concept StateInfo(GetStateRetType, Memento);
  concept Reset(Originator, Memento, Originator);
  concept Restored(Originator, Originator);

  lead role Originator state {}  
  role Memento state {}  
  adjunct Memento state { Originator _savedO; }  

  simple role GetStateRetType;

  constraint:
  \[ \text{SameMem : reflexive, transitive;} \]
  \[ \text{Restored : reflexive;} \]
  \[ (\forall \text{ Originator o1, Memento m1, Memento m2 ::}
    \quad (\text{MemCopy(o1, m1)} \land \text{SameMem(m1, m2)}) \Rightarrow \text{MemCopy(o1, m2)}) ; \]
  \[ (\forall \text{ Originator o1, Originator o2, Originator o3, Memento m1 ::}
    \quad (\text{MemCopy(o1, m1)} \land \text{Reset(o2, m1, o3)}) \Rightarrow \text{Restored(o1, o3)}) ; \]

  instantiation: Originator.new() {
    lead: target;
    ensures: true
  }

  invariant:
  \[ \text{instance.Originator = \{ instance.lead } \]
  \[ \land (\forall \text{ Memento m } \in \text{instance.Memento :: MemCopy(m._savedO, m)}) \]

  \ldots role contracts for \text{Originator} and \text{Memento}, see Figs. 5.11 and 5.12 \ldots
}

Figure 5.10: Memento pattern contract
the originator. As will become more clear when we go through the method contracts, explicitly representing this part of the state is unnecessary due to what the given auxiliary concepts express. In a sense, this state information is internalized by the auxiliary concepts.

However, we still must address how we can refer to the original originator state that is supposed to be encoded in the memento. Recall that the state maps in subcontracts are used to project parts of the role state out from the current concrete state of the role object; a correct role mapping is not responsible for meeting the requirements imposed on the role object itself. To illustrate the problem, suppose we have a memento whose internal state changes such that a substantially different originator would result from the original when SetMemento() was invoked. We may be able to extract the originator state currently stored in the memento using a state map, but how would we be able to tell if this is the originator state that should be stored? This problem is taken care of using an adjunct state variable _savedO for the Memento role that does store this information about the original originator state. Recall that adjunct state variables do not have to be part of the internal state of the role objects, and so are not given definitions in the subcontracts as are the role state variables. Instead, these adjunct variables' values are set at certain points, as specified in the pattern contract. Here, the _savedO variable is set when CreateMemento() is called, as specified in Fig. 5.12.

Next, we define the simple role, GetSateRetType, which simply acts as a stand-in for the actual type of the objects returned by the GetState() method. Since different applications of the Memento pattern may return objects of a different type, the actual concrete type will be specified in the subcontract. Moving on to the instantiation
lead role Originator contract {

Memento CreateMemento():
   ensures: MemCopy(#this, result)

SetMemento(Memento m):
   requires: m ∈ instance.Memento
   ensures: Reset(#this, m, this) ∧ Restored(m._savedO, this)
             ∧ MemCopy(m._savedO, m) ∧ SameMem(#m, m)
   others:
       ensures: true
}

Figure 5.11: Originator role contract

clause, we see that it simply states that a new pattern instance is created when a originator is created, which will be the lead object for the instance. Next, we have the invariant, which first states that the single originator in the pattern instance is the lead, and then that all of the mementos’ internal states faithfully represent the value of the originator at the time they were created. Note that if _savedO was a role state variable and represented the current originator state saved in the memento, this would not convey the full intent of the pattern.

Now, we will look at the Originator role contract in Fig. 5.11. In the specification for CreateMemento() here, we only stipulate that the produced result be consistent—as defined by MemCopy()—with the originator. The enrollment-specific specifications for CreateMemento() are found later in Fig. 5.12. The next method, SetMemento(), first requires that m be created from this originator. Because originators do not keep track of their own mementos, we instead make use of the instance keyword here to
role Memento contract {
    enrollment: Memento Originator.CreateMemento() {
        lead: target;
        enrollee: result;
        ensures: true
        set adjunct: result.savedO = StateCopy(target) }
    GetStateRetType GetState():
        requires: caller == instance.lead
        ensures: StateInfo(result, this) ∧ MemCopy(this._savedO, this)
        ∧ SameMem(#this, this)
        others:
            ensures: MemCopy(this._savedO, this) ∧ SameMem(#this, this)
    }

    Figure 5.12: Memento role contract

obtain this set of mementos associated with the pattern instance\(^{23}\). The ensures clause
first states that SetMemento() Reset the originator’s value as expected, and that the
value of the original state (as saved in _savedO) is sufficiently similar, as defined by
Restored(). The next two conjuncts then go on to say that the memento’s state is
consistent with the original originator state, and that its state was not significantly
changed.

Moving to the Memento role contract in Fig. 5.12, we first have the enrollment
clause, which states that a memento enrolls by virtue of being returned by the
CreateMemento() method. The set adjunct clause sets the value of _savedO after

\(^{23}\)An alternative would be to use another adjunct variable for Originator that stored these me-
mentos. However, it would always contain the same values as instance.Memento since it would be
updated whenever a memento was enrolled, and so we opted not to use it.
CreateMemento()’s execution. Here, StateCopy() is a function that creates a deep-copy of the argument’s concrete application state\textsuperscript{24}. The next method specified is GetState(), which returns an object that represents the originator state stored in the memento. The return type is given as GetStateRetType here in the contract, which can be mapped to a concrete type for a particular application in the subcontract. To help specify the correct output of this method for a specific system, the StateInfo() concept has to be appropriately defined in the subcontract. Along with this condition, we also use MemCopy() and SameMem() to ensure that the memento remains consistent with the original state, and that it has not significantly changed.

5.4.4 Chain of Responsibility

The intent of Chain of Responsibility \cite{33} is to decouple the handler of a request from the requestor, so that multiple handlers have a chance to service it. When a request is made of a handler, the handler determines whether it can service the request. If possible, it does so; otherwise, it forwards the request to another handler that is linked to the current handler. Each handler object can have one such successor handler linked to it that will receive its forwarded requests. One constraint on the topology is that there are no cycles resulting from the successor relation, which could lead to requests being indefinitely passed around among a group of handlers. This leaves us with one or more directed tree graphs whose root nodes are reachable from all other nodes in the tree.

To simplify the writing of the contract, while not placing constraints on the topology of the successor graph, we define each instance of the pattern to be a single

\textsuperscript{24}For runtime monitoring purposes, the MonGen tool generates methods for saving state that attempt to do deep-cloning by inspecting the fields of classes playing roles in the system. We will discuss this further in the next chapter.
handler. In this way, each node of the graph is itself an instance of Chain of Responsibility, and each non-null successor refers to a handler that constitutes another pattern instance. PCL places no restriction on interactions among objects of different pattern instances. An alternative definition of instance here that grouped together several linked handlers would have to address difficulties such as how to group the handlers together into instances (and what if we allow handlers to participate in separate instances), determining who would play the lead in each instance, and defining the instantiation and enrollment clauses in such a way as to handle all of the bookkeeping involved in maintaining these instances.

The PCL specification for Chain of Responsibility is given in Fig. 5.13. In the contract, we use the concept CanHandle() to capture the notion of whether a particular handler can directly service a request. Handler is the only (non-simple) role in the pattern, and has the single state variable succ to store its successor handler. To accommodate the types used for requests and responses, we just use simple roles. The CanHandle() concept does not have any constraints. The instantiation clause simply states that a new pattern instance is created for each new handler. The invariant is primarily concerned with preventing cycles. First, we simply state that the lead is the only member of the instance. Then, we use a special definition SafeLoopCount, which essentially counts the number of times we would iterate through the for loop with the given header, but returns −1 if the same index is encountered more than once. This is handled similarly to the special definitions and quantifications discussed in Sec. 3.3.1.

The next part of the specification covers the role method contract for Handler. The named method HandleReq() handles the request given to the handler, and returns a
pattern Chain_of_Responsibility contract {
  concept CanHandle(Handler, Request);

  lead role Handler state { Handler _succ; } 

  simple role Request;
  simple role Response;

  constraint: true;

  instantiation: Handler.new() {
    lead: target;
    ensures: true }

  invariant:
  instance.Handler = { instance.lead }
  ∧ SafeLoopCount(Handler h = instance.lead; h ≠ null; h = h._succ) ≠ −1

  lead role Handler contract {
    Response HandleReq(Request req):
      requires: ¬CanHandle(this, req) ⇒ _succ ≠ null
      preserves: _succ
      ensures: (CanHandle(this, req) ⇒ |τ| = 0)
      ∧ (¬CanHandle(this, req) ⇒ (|τ| = 1 ∧ τ._succ.HandleReq = 1
          τ[0].arg1 == req ∧ τ[0].result == result))

      others:
      ensures: true
  }
}

Figure 5.13: Chain of Responsibility pattern contract
response. Its requires clause makes sure that if a request cannot be directly handled by the handler that it does have a successor to which it can forward the request. The preserves clause states that the successor is not changed by the method. The first conjunct of the preserves clause says that if the handler can handle the request, then it should not make any other calls to a Handler role method. The second conjunct describes what happens if the request cannot be directly handled by the handler: a single call is made, which is the forwarding of the request to the successor _succ, and the result of the forwarded call will be returned by the method. No constraints are put on the non-named methods (other than the invariant), since these methods may be used to manipulate the successor of the handler.

5.5 Behavioral v. Non-Behavioral Patterns

One key question here is whether our formalism is sufficiently powerful to enable us to write precise contracts for the various design patterns that have been discovered and documented in the literature. The patterns that we have presented here, Observer, Memento, and Chain of Responsibility, are classified in the pattern literature as behavioral patterns. So one particular question would be whether the approach is capable of handling other classes of patterns, i.e., structural patterns and creational patterns. We believe the answer is that even patterns in these classes have some behavioral aspects associated with them. For example, the Builder pattern which is usually classified as a creational pattern includes a role, Director, that is required to have an operation that has a structure somewhat similar to that of Notify() of the Subject role. This requirement can, of course, be captured using traces in the same
manner as in our Observer contract. Similarly, the Composite role of the Composite pattern (which is generally classified as a structural pattern), is required to have a method that invokes a certain specified operation on each of the Component objects that it contains. Again this is a behavioral aspect and can be captured in the corresponding contract.

Some aspects of these patterns are entirely structural and are best expressed using UML diagrams or other syntactic descriptions against which a system using the pattern can be statically validated. These, however, are not behavioral requirements that have to be monitored at runtime. To put it differently, most patterns have both structural and behavioral aspects to them. The former may be specified using suitable graphical or other syntactic descriptions, and conformance of systems against these can be completely checked by syntactic inspection either manually or by suitable static analysis tools. The latter, involving behavioral considerations, would have to be specified using an approach such as ours.

5.6 Related Work

Helm, Holland, and Gangopadhyay’s [39] contract formalism shares some similarities with ours. For example, their formalism provides a construct similar to our auxiliary concepts. It does not, however, provide a way to impose constraints on these definitions that may be supplied. The formalism also includes support for specifying the relative order of method invocations, but the constructs are underdeveloped. It is impossible, for example, to quantify over a method call sequence to require that a particular method be invoked exactly once, or alternatively, that a particular method
not be invoked at all. Finally, there is nothing analogous to the others clause to prevent the unnamed methods from violating a pattern’s intent.

We are not the first to consider pattern formalization. Eden [26, 27] proposes a higher-order logic formalism in which patterns are expressed as formulae. Each formula consists of a declaration of the participating classes, methods, and inheritance hierarchies, and a conjunctive statement of the relations among them. While rich structural properties can be expressed, there is limited support for behavioral properties. The formalism does not, for example, provide constructs for referring to pre- and postconditional values, nor does it provide a concept analogous to our method call sequences. In contrast, Mikkonen’s approach [61] focuses almost exclusively on behavioral properties. In his approach, patterns are specified using an action system notation. Data classes model pattern participants, and guarded actions model their interactions. The approach is well-suited to reasoning about temporal properties. One limitation, however, is that the separation of actions and data is structurally inconsistent with the OO paradigm, making it difficult to express most structural properties. Further, Mikkonen’s specifications cannot be specialized to the needs of particular systems; thus pattern flexibility may be seriously compromised. Soundarajan and Hallstrom consider pattern formalization in [80], and lay some of the groundwork for this work. However, they do not present a general pattern specification language, nor do they consider pattern specializations. Furthermore, they do not consider the automated generation of monitoring code.
MONITORING DESIGN PATTERN CONTRACTS

In the previous chapter, we discussed how design patterns can be specified in a precise way, while retaining the flexibility inherent in the pattern. We presented the specification language PCL that is able to capture the properties common across all applications of a pattern, while accommodating the variation that occurs across those applications. In this chapter, we turn to how PCL specifications can be converted into runtime monitors, which determine whether the specifications are respected in a given system. We present the MonGen tool which automates this process, and illustrate its use on several examples.

6.1 Overview of Contract Monitor Generation

MonGen\textsuperscript{25} generates monitoring AspectJ \cite{aspectj} aspects from parsing PCL contracts for a set of patterns and subcontracts corresponding to the specializations of the patterns in a given system $S$. These aspects produced by MonGen can then be woven together with $S$ by the AspectJ weaving compiler to produce a system that will behave in the same way as $S$ when executed, except that it will monitor the system.

\textsuperscript{25}MonGen was developed using the ANTLR parser generator \cite{antlr}, and is an open source system.
Whenever the requirements specified in the pattern contracts as specialized in the subcontracts for $S$ are violated, appropriate messages are output by the monitor.

These aspects are valuable not only during the initial stages of the development process, but throughout a system’s lifecycle. These monitoring aspects can be reused during periods of evolution and maintenance, to check whether subsequent changes made to the system violate the intended design of the system as specified in the pattern contracts.

In Sec. 6.2, we will illustrate the process by which MonGen generates these aspects showing the flow of attributes in the PCL parse trees. Before we can go into too much detail about how these aspects are generated and how they work, we should first describe the key parts of the AspectJ aspects that are used by the contract monitors. A join point identifies particular places in the execution of a program where we may wish to add to or otherwise change the program’s behavior. In the current version of MonGen, the joint points are always at method executions. A pointcut groups together a set of join points that we want to treat in a uniform manner. Pointcuts enable us to collect context, for example, the object on which the method in question was applied, parameter values, etc. Finally, the advice associated with a pointcut specifies the code that should be executed when control reaches any of the join points that match the pointcut. There are three kinds of advice. Suppose a join point is a method call. The before advice, if any, is executed before the method is executed, the after advice after the method is executed. We do not use third kind, around advice. Like classes, aspects can also contain their own local state and methods.

The flexibility that is gained in separating the subcontracts from the pattern contracts is reflected in the contract and subcontract aspects generated by MonGen.
The generated contract aspects are *abstract*, and do not contain concrete definitions for things that are to be specialized by the subcontract. Subcontract aspects, or *subaspects* as we will often refer to them, which extend the concrete aspects provide these definitions so that the specific application’s behavior relative to the contract can be monitored.

### 6.1.1 Aspect and Subaspect Responsibilities

Given a pattern contract, for each *named* method of each role, a pointcut as well as a block of *before* and *after* advice associated with that pointcut are created. The *before* advice checks the precondition specified in the contract for the particular *named* method while the *after* advice checks the corresponding postcondition. Although the contracts give specifications for the role methods, the applications are not constrained to have methods with the same names and parameters. Thus, these pointcuts are defined as abstract, and will be provided concrete definitions (where method names and arguments will be mapped to those given in the role methods) when a corresponding subcontract aspect is generated later. A more important source of flexibility, the auxiliary concepts, are handled in a similar fashion where MonGen introduces an abstract (boolean-valued) function in the contract aspect for each. A third source of flexibility are *role maps*, which allow us to convert the role state into the application state. Again, abstract methods are used in the contract aspect for these, and are given concrete definitions in the subcontract aspect. A summary of where the PCL elements are handled in the monitoring aspects is shown in the table in Fig. 6.1.
6.1.2 Multiple Pattern Instances

As we have mentioned, several instances of the pattern $P$ may be created and used during a particular execution of a system $S$, and MonGen was designed to be able to simultaneously store information about all the instances that are in existence at any given point. Within the contract aspect, MonGen creates a $\text{PatternInstance}$ class, which contains a $\text{HashSet}$ for each (non-simple) role defined in the contract to contain those role objects, plus a $\text{lead}$ variable that is the lead object of the instance. The $\text{lead}$ variable is given the type of the role that is flagged as the lead role in the contract.

To keep track of all of the instances within the system, the contract aspect generated by MonGen contains a $\text{playerMap}$ and a $\text{patInstMap}$. The $\text{playerMap}$ maps each of the objects that play some role in some instance of the pattern, and returns the
Set of PatternInstances in which it is involved. The patInstMap maps each of the lead objects to the pattern instance that they denote. These mappings are updated whenever an instance is instantiated (or deinstantiated), or an object becomes enrolled (or disenrolled).

In the PCL contracts, the specifications for the role methods are stated relative to a single instance of the pattern being specified, although it is reasonable to assume that a single application object can participate in multiple pattern instances. As we have mentioned before, when an application method corresponding to a role method is invoked, the method pre- and postconditions, and the preserves and adjunct state setter clauses are evaluated relative to each pattern instance to which the object belongs. As we will see later, this is implemented in the advice code with a loop that iterates through all of the pattern instances that the object is involved with, and evaluates the clauses relative to the variables instance and lead that are set accordingly for each iteration. Recall that the variable lead can be used within the role map definitions in the subcontracts, so that the role state variables can be evaluated relative to a given pattern instance. Note that in the enrollment and instantiation clauses, however, we specifically know what pattern instance is under consideration, so we only have to evaluate those once.

6.1.3 Saving Role State and Traces

MonGen also has to keep track of the trace for each method as they are executed, since the specification of the method may refer to it. The before advice for each method m() of each role (including the unnamed methods of the role) creates and initializes an empty trace which will be used to record the methods that m() calls.
When \( m() \) finishes execution, its \textit{after} advice can use this trace to check that the conditions listed in \( m() \)'s postcondition are satisfied. Consider a call that a method \( n() \) makes to the method \( m() \). The \textit{before} advice for \( m() \) creates and initializes the new trace for this execution of \( m() \) and pushes it on the \texttt{TraceStack}; then, its \textit{after} advice pops this trace off of the \texttt{TraceStack} to do the postcondition evaluation, and then updates the trace of \( n() \) (which is now at the top of \texttt{TraceStack}) to record this call to \( m() \) made by \( n() \). This is essentially the same procedure used in the monitors generated by TCGen described in Chapter 3. To keep track of the pre-state of the method arguments that may be mentioned in postconditions and are needed for evaluating preserves clauses, the monitors store the state in the \texttt{caller TraceRec} variable of the \texttt{Trace}. MonGen also creates specialized \texttt{TraceRecs}, as does TCGen.

However, there is a slight difference in how MonGen and TCGen save state. Where TCGen only uses a one-deep copy mechanism, MonGen does a field by field reference copy for non-container fields of the application class, and for the container fields of the class, it copies all of the elements' references over into a new container object field. This slightly deeper copy is used in an effort to save enough application state so that the intended role state information can be extracted from it—the one-deep copy mechanism is generally not sufficient in practice. Given that the overall state structure may vary from application to application, it is still possible that the cloning methods implemented by MonGen in the subaspects are not sufficient for certain cases. These methods, however, can be rewritten by the developer/tester to ensure that the proper state information is saved to allow for the intended role values to be produced by the subcontract \((\texttt{varMap})\) code.
Using new objects from the application classes to hold onto these values could also be problematic because they could be active objects in the system, which may cause the system to behave differently than it would if no such new objects were created at runtime. MonGen handles these problems by generating *shadow classes* for each of the application classes mentioned in the subcontact. Shadow classes contain the same internal data members as the application classes (this information is extracted using Java reflection), and these values are copied over using the above procedure. These shadow classes are also made to implement the dummy interfaces (which act as stand-ins for the role types), so their objects can also be used directly when evaluating the specifications in the abstract aspect, and can be saved in the trace records defined in the abstract aspect.

6.2 Contract Monitor Generation Details

6.2.1 Pattern Contract Aspects

Now, let us start by looking at the overall parse tree for ⟨patternContract⟩ shown in Fig. 6.2, which is generated by the contract grammar in Figs. 5.1 and 5.2. In our parse tree representations in this chapter, we will represent the fact that multiple branches having the same kind of nonterminal node can be a child of another nonterminal by using an oval with an ellipsis inside on a branch going from one to the other. For example, the ⟨auxConcepts⟩ node can have multiple ⟨auxConcept⟩ nodes as children, and so we put the oval between them in the diagram. Also, in our parse trees, we will sometimes show some of the terminal tokens to help remind us what part some of the nonterminals play in generating the attributes. In the tree for ⟨patternContract⟩, note that we have omitted the ⟨constraint⟩ nonterminal; this is because constraints
are generally not checkable, and so are not processed by MonGen. We have also left off \(\langle\text{deinstantiation}\rangle\) and \(\langle\text{disenrollment}\rangle\). The rules used to handle these (and their associated attributes) are very similar to those for \(\langle\text{instantiation}\rangle\) and \(\langle\text{enrollment}\rangle\), respectively, and so they are also omitted.

Let us move on to Fig. 6.3, which shows the rule for \(\langle\text{auxConcepts}\rangle\). Here, we encounter our first attribute, CAMCODE. All attributes will be shown in uppercase, sans serif font in our diagrams, and their definitions will appear in brackets. The “direction” in which the attribute will go in the parse tree will be shown with arrows. Here, CAMCODE gives the code for the abstract method to be placed in the contract aspect for the given auxiliary concept. The name of the method is simply the same as the name of the concept (\(\langle\text{cid}\rangle\)). The method’s arguments, which need to have both
types and parameter names, are generated using the `typArgs()` function\(^{26}\); \(\langle rids \rangle\) only lists types, and so parameter names need to be created. As the `CAMCODE` attributes flow up the parse tree to the root (\(\langle \text{patternContract} \rangle\)), they are collected together at the oval with the ellipsis, so that we will have all of the `CAMCODE` definitions at the root. Other attributes that flow through such an oval are handled the same.

Next, we go to the subtree for \(\langle \text{roleStates} \rangle\) (Fig. 6.4). Here, we have the inherited attribute \texttt{LEAD}, which comes from the root—this stores the name of the lead role, \(\texttt{LEAD}\),

\(^{26}\)We use sans serif font for actual code, and Times font for functions that take attribute values, and convert them to code.
which is determined when the lead keyword is encountered during parsing of the role information. First, we generate the abstract method code for the role state that will appear in the contract aspect, and store it in the RSAMCODE attribute. Again, role state variables are not necessarily part of the actual application classes, so a mapping function has to be provided here—it's concrete definition will be defined in the subcontract. Also, recall that because the value of the role state may depend on which pattern instance we are observing, we disambiguate using the lead object of the pattern instance in question; this second parameter is used to take the given lead object. The RSAMCODE attributes will be collected together at the ovals when flowing to the root node of the tree, so that all of these abstract methods will be part of the contract aspect. A second attribute, RS, collects the basic information here: that the given role ⟨rid⟩ has a role field ⟨sid⟩ that has the type ⟨typeName⟩. This RS information is later used when processing pre- and postconditions, preserves, and adjunct state setting clauses, so that they can be converted from the field access notation used in the specifications, into method calls in the actual evaluation code.

The subtree for ⟨adjStates⟩ is shown in Fig. 6.5. Although the syntax for adjunct state is similar to that for role state, they are handled differently. Adjunct state is defined at the contract level, and not at the subcontract level, and so is handled by the contract monitoring aspect. For each adjunct state variable, a special HashMap called ⟨sid⟩_map is generated which holds onto the adjunct state values for each of the objects playing the given role. However, as is the case with the regular role state variables, we may need to disambiguate using a lead object. Thus, the mapping ⟨sid⟩_map is actually a mapping from the role objects to another mapping. This second mapping is from the possible lead objects to the actual value of the adjunct
state variable. For clarity, we leave out the type casting code, as we will for the rest of the discussion. The second attribute which is passed up the parse tree, \texttt{AS}, serves the same function as the \texttt{RS} attributes for the role state.

For the \texttt{<simpleRoles>}, shown in Fig. 6.6, very little needs to be done in the aspect code. As with the non-simple roles, the simple roles should be defined as actual types in the monitoring system. This is done by defining a “dummy” interface for the simple role in \texttt{SRICODE}, which will be part of the contract aspect code. This is
necessary in order for us to define parameters and variables in the contract aspect as being of such a type. Unlike the non-simple roles, however, little else needs to be done for the simple roles.

Next on the parse tree is the \textit{instantiation} rule, whose parse tree is shown in Fig. 6.7. For the instantiation clause, separate pointcut and advice code are \textit{not} created. Instead, the instantiation information is processed and inserted into the advice code of the named method which matches that specified in \textit{roleMethSig}. This instantiation information is all stored in the \texttt{INST} attribute, which has three fields: \texttt{SIG}, which holds the method’s signature used for matching with the named method; \texttt{IECODE}, which is the actual code used in the advice to create a new pattern instance (done via a simple method call); and \texttt{XCODE}, which holds onto the executable evaluation code obtained by processing \textit{reqPresEns}. The \texttt{XCODE} attribute itself
Figure 6.8: Parse subtree for \(\langle reqPresEns \rangle\)

has four fields, REQ, PRES, ENS, and ADJ, which holds the evaluation code for the requires, preserves, ensures, and set adjunct clauses, respectively. Before explaining how \texttt{INST.XCODE} is being defined here, we should first go over how we arrive at the \texttt{XCODE} attribute for \(\langle reqPresEns \rangle\) itself.

Here, in Fig. 6.8, we show the subtree for \(\langle reqPresEns \rangle\), and its synthesized \texttt{XCODE} attribute. The function \texttt{EvalCode()} is used to process the requires and ensures clauses, and uses the same transformation mechanisms used in TCGen to handle pre-state variables, special expressions, quantifications, etc. Furthermore, references to both the role fields and adjunct fields—which will use the standard dot notation for field references—need to be transformed into the proper method calls. The RS and AS
attributes, which were collected while processing ⟨roleStates⟩ and ⟨adjStates⟩, along with the typing information for the method’s parameters, which is passed in via the VARMAPS attribute, are used to locate these field references that need to be converted in the expressions. The PresCode() function is used to convert the preserves list into evaluation code, where we use “element equality” for container objects, and reference equality for non-container objects. Because role and adjunct state variables can also be listed in the preserves clause, the RS and AS attributes (along with VARMAPS) is used to convert these field accesses to method calls.

The code used to handle the setting of adjunct state in the ADJ field is somewhat symmetric to that found in Fig. 6.5, where we retrieve the adjunct state in the ASAMCODE attribute. Here, though, we must first evaluate the expression to which the adjunct state “variable” will be set; this is done by again using the EvalCode() function to convert the ⟨valExpr⟩ into executable code27. After that, the map for the given object is then fetched from the ⟨aid⟩map, where the new value is associated with the current lead variable. (We will see later that this variable lead will be set appropriately before this code is executed.) Then, this secondary map is placed back into the ⟨aid⟩map.

Returning to Fig. 6.7 for the ⟨instantiation⟩ subtree, we see that the XCODE for the requires clause remains the same, whereas the XCODE for the preserves, ensures, and adjunct state setting clauses have code prepended to them to fix the lead and instance variables to the current lead object lid and its associated pattern instance.

27Here, to make the presentation more clear, we treat the code produced by functions such as EvalCode() as if it is a single expression. In reality, the evaluation code often will be made up of several statements, such as when we have quantifications. What is really produced is a code block that is to be inserted right before the assignment, where the block assigns a value to a given variable that acts as a sort of “handle”, which is then used on the right hand side of this assignment.
These variables are fixed so that they are evaluated relative to this specific pattern instance. This is not done for the requires clause, since this instance does not exist before the call (and thus cannot be referenced in the requires clause either). The variable patInstMap is a variable of the contract aspect, and holds a mapping from each lead object to the particular pattern instance that it instantiated—this variable is updated in the code for addInstance() (as well as in the code for removeInstance(), which is called during deinstantiations).

Moving on to Fig. 6.9, we come to the subtree for the \langle invariant \rangle rule. Here, the method checkInv() is constructed, and passed up the parse tree in the INVARCODE attribute. The checkInv() method is contained in the contract aspect, and checks to see if the invariant as specified in \langle invExpr \rangle is violated for any of the pattern instances in which the object obj plays a role. We will see shortly that this is invoked at the beginning and end of each method that is called from an external source (i.e., when
the TraceStack is empty), where the argument is the object on which the method was called. The playerMap variable is another bookkeeping variable contained in the contract aspect that returns the set of pattern instances that the passed argument is involved with, and it is updated at each instantiation, deinstantiation, enrollment, and disenrollment. Note that the second parameter to EvalCode() is empty—this is where the VARMAPS go, but since the invariant is not specific to any method, there are no such variable mappings.

Next, we turn to the ⟨roleContracts⟩ rule. Here, for each ⟨roleContract⟩, we generate a dummy interface as we did for the simple roles, and send them up the parse tree to be placed in the contract aspect. Each ⟨enrollment⟩ is handled in a virtually identical way as the ⟨instantiation⟩ rule, and so we omit its parse tree diagram. Again, no pointcuts or advice are generated; their processed code is forwarded to the proper named methods in the parse tree (via the ENROL attribute), where they are inserted in their advice. The only notable difference from ⟨instantiation⟩ in the generated evaluation code XCODE for ⟨enrollment⟩, is that it also fixes the pattern instance and lead object for the requires clause; in this case, the pattern instance can exist before the enrollment occurs.

Moving on to the subtree for the ⟨namedMethodSpec⟩ rule, we will first examine the NMSIG attribute, and then how pointcuts are generated (Fig. 6.10). First, the VARMAPS are generated, which includes the typing information for the method’s role object, as well as a possible return value. This is then used to create the NMSIG attribute, which is passed on to the root. This information is used to construct the specialized TraceRec classes for the role methods, which are very similar to those created by TCGen for monitoring frameworks. The information in the NMSIG attribute
is also passed on to the subcontract parser, to aid in creating the concrete pointcuts in the subcontract aspect. Turning to pointcuts, we first note that they are named using the name of the role, then an underscore, than the role method’s name. In the case where the role method is not a constructor, an abstract pointcut is created, which is shown in the diagram. The pointcut will be later defined in the subcontracts, which essentially will map the given application class methods to the role methods specified in the pattern contract. If, however, the named method is a constructor, i.e., the \( \langle \text{methId} \rangle \) is \texttt{new}, then we can create a concrete pointcut here, defining it as being the execution of the constructor method. The last attribute here, \texttt{PCNEG}, is used by the parent \( \langle \text{roleContract} \rangle \) to create the pointcut for the unnamed methods of the role.

We now turn to the generation of advice for each \( \langle \text{namedMethodSpec} \rangle \). First, we define the attributes \texttt{NMIEXCODE} and \texttt{NMIEXCODE} solely as a convenience for advice generation. The \texttt{NMIEXCODE} is simply the enrolling and instantiation code.
associated with this named method—these will be simple invocations of the methods enroll() and addInstance() defined in the contract aspect. Likewise, the NMIEXCODE attribute contains the requires, preserves, ensures, and adjunct state setting clauses defined in the instantiation/enrollment clause whose signature matched that of this method. For advice generation, we also have the XCODE attribute generated from the ⟨reqPresEns⟩ nonterminal for this named method.

In Fig. 6.11, we show how the before advice is created from these attributes. The first line itself just associates the advice with the pointcut. The next line, the first line of the advice body, checks the invariant on the role object this (on each of the pattern instances it plays a role in) if this call was made externally relative to the monitored part of the system. This is determined by checking if the height of the

Figure 6.11: Creation of before advice
TraceStack is zero. The next three lines, which are grouped together in a box, checks the requires clause of the instantiation or enrollment clause, if there is one associated with this named method. The first line simply declares the variables patternInst and lead, which will be set by the evaluation code for the requires clause here. After the requires clause is evaluated, a message is printed to the screen if it is violated.

The next box handles the checking of the regular precondition—the precondition obtained from the requires clause defined in the named method specification—relative to each of the pattern instances in which the role object plays a part. The first line simply declares a boolean to be used in these evaluations; the second line obtains an iterator to the pattern instances in which the object plays a part. Inside the while loop, an instance is obtained from the iterator, and the lead object defined. Defining the lead here is crucial; it is needed to disambiguate when the role state methods are invoked. Recall that the second parameter to these methods were defined specifically for this purpose. The next line handles the evaluation of the precondition (relative to the current lead), and then the one after prints a message if it is violated.

The next two lines of the advice, which are past the second box, creates a new Trace object using a specialized TraceRec for this method as its caller. The final line of the advice pushes it on the TraceStack. It should be noted here that the pre-state for that object and parameters that will be used in evaluating the preserves and ensures clauses later are saved here in the trace. Unlike TCGen, MonGen generates deep copy operations for the application objects within the subaspect code, which are used to save state in the specialized TraceRecs.

For the generation of after advice, shown in Fig. 6.12, we first associate this advice code with the proper pointcut, and we also include a returning clause if the
AFTADVICE

after (RID _this, Args(<args>)) returning (<retType> result): 
    RID_<methId>_this, Params(<args>) { 
    PatternInstance instance; LEAD lead; 
    NMIEXCODE; 
    Trace tau = TraceStack.pop(); 
    RID_<methId>_TraceRec caller = tau.caller; 
    caller.saveResult(result); 
    RID _pre_this = caller._pre_this; 
    GetPreArgValues(<args>, caller); 
    boolean iepres_res = NMIEXCODE.PRES; 
    if(! iepres_res) System.out.println("Inst/Enroll preserves clause violated!"); 
    boolean ieens_res = NMIEXCODE.ENS; 
    if(! ieens_res) System.out.println("Inst/Enroll postcondition violated!"); 
    boolean pres_res, ens_res; 
    Iterator it = playerMap.get(_this).iterator(); 
    while(it.hasNext()) { 
        instance = it.next(); lead = instance.lead; 
        XCODE; 
        pres_res = XCODE.PRES; 
        if(! pres_res) System.out.println("Preserves clause violated!"); 
        ens_res = XCODE.ENS; 
        if(! ens_res) System.out.println("Postcondition violated!"); 
    } 
    TraceStack.appendToTopTrace(caller); 
    if (TraceStack.height() == 0) checkInv(_this); 
} 

Figure 6.12: Creation of after advice
role method is to return a value. The first line shown in the advice body is the **NMIECODE** attribute—this is the code that does the enrolling/instantiating by invoking the proper methods of the contract aspect. The next block of code, which is inside the box, first retrieves the current trace \texttt{tau} from the \texttt{TraceStack}. The trace \texttt{tau} is not only used for postcondition evaluation, but it also holds the pre-state that was saved at the end of the corresponding \texttt{before} advice. First, the \texttt{caller} specialized \texttt{TraceRec} is referenced from \texttt{tau}, then we save the \texttt{result} from the method invocation if the method returns a value. We proceed to extract the pre-state of the role object from \texttt{caller}, and then do the same for the method’s parameters, illustrated by the function \texttt{GetPreArgValues()} which generates the proper code for each argument.

The next block of code, shown in the second box, takes care of setting the adjunct variable, and evaluating the preserves and ensures clauses for the (possible) enrollment/instantiation associated with this named method. First, though, we declare the \texttt{instance} and \texttt{lead} variables, which will be set to the proper values in the **NMIEXCODE**. The third boxed block of code does the same thing, except the \texttt{XCODE} here (taken from \langle \texttt{reqPresEns} \rangle for this named method) is evaluated relative to each pattern instance in which the role object plays a part. An iterator \texttt{it} is used to go through each of these instances. Within the body of the loop, the adjunct state variable is set, the preserves clause is checked, and then the postcondition is checked.

At the end of the \texttt{after} advice, we append the record of the current call, \texttt{caller}, to the current trace found on top of the \texttt{TraceStack}. Then, we check the invariant if this call was made external to the monitored part of the system.

We now move on to the \texttt{othersSpec} for the given \texttt{roleContract} role. The creation of the pointcut here is straightforward. This pointcut is not abstract in the contract
aspect, as are the (non-constructor) pointcuts, because it can be defined as all of the method executions on an object of this role that are not captured in the pointcuts for the named methods. This is where the PCNEG attribute for this role is used. The advice code for the unnamed methods is similar to that of the named methods, except that there is no need to deal with associated enrollment/instantiation code. This is because such code can only be defined in the context of a named method.

6.2.2 Subcontracts and Subaspects

The second task of MonGen is to generate suitable subcontract subaspects, given the subcontracts corresponding to the specializations of the various patterns used in a system \( S \). The subaspect that MonGen produces corresponding to the specialization of a particular pattern \( P \) will be an extension of the contract aspect, and will provide definitions for the abstract pointcuts as well as the abstract methods in the aspect that MonGen produced from \( P \)'s pattern contract.

We will start by looking at the parse tree for \( \langle subContract \rangle \) in Fig. 6.13. The
first task that MonGen handles is to force the application classes to implement the dummy interfaces that were created during the contract aspect generation. This is done using the declare parents construct provided by AspectJ, where the code itself is generated when the \(cid\) as \(srid\) and \(cid\) as \(rid\) parts of the \(simpleRoleMap\) and \(roleMap\) subtrees are parsed.

Looking down the subtree for \(roleMap\), the \(stateMap\) and \(interfaceMap\) will also be processed by MonGen. Before we look at the diagram, recall that in the contract aspect, many of the argument types that are used are roles. Now, if several different application classes play the same role, we cannot simply make a single method or pointcut for each of these different classes—we are still required to define our methods and pointcuts using the same signature as they have in the contract aspect. What we need to do, then, is to create several different cases in the method bodies (making use of the instanceof operator in Java), or several different disjuncts in the pointcut definition, each pertaining to an application class playing that role.

Since much of the code creation is done at the root level of the parse tree, many of the other rules simply propagate the information they contain up the parse tree using attributes.

Starting with the \(stateMap\) subtree, shown in Fig. 6.14, we see how fragments of the role state method definition are constructed. Again, in the contract aspect, abstract methods were declared that mapped a role object, lead object pair to a value for the role state field. We cannot construct the entire concrete method definition at this depth of the parse tree, because other classes may be mapped to the same role, and will provide their own return code for the role state variable. Instead, a code fragment of the method is produced here that first checks the type of the object, and
executes the \textit{returnCode} if there is a match. When the parsing of the \textit{roleMaps} are complete, the individual \texttt{RMAPFRAG} attributes are grouped together by method name (\texttt{METH}) and role type (\texttt{ROLE}), and each groups’ code is appended together to form the body of the role state method. A very similar process is used when parsing the \texttt{methodMaps}, where the information about what application methods for the given class match up with the role methods outlined in the contract. Pointcut fragments are created here, and flow up the parse tree to the root, where they are grouped together and appended accordingly to create the concrete pointcut definitions for the subaspect.

The \textit{auxConceptDefs} are also handled in a similar fashion, as seen in Fig. 6.15. Again, we use the \texttt{instanceof} operator to match the parameters to their given types; in the diagram, we use shorthand here to mean that the \texttt{instanceof} operator is applied
for each of the arguments, and all of the conditions are conjoined together in the code as the condition for the if-statement. The produced fragments are again passed up the parse tree where they are grouped and assembled accordingly to create concrete definitions for the auxiliary concept methods in the subaspect code.

6.3 Example Monitoring Code

We will now consider the abstract aspect generated from the Observer contract shown in the previous chapter (see Fig. 6.16). The aspect begins by declaring interfaces for each of the roles defined in the pattern contract. These interfaces are mapped to the appropriate system classes in the subaspect based on the role maps included in the subcontract. The subaspect generated from the Hospital subcontract, for example, maps the Subject interface to the Patient class using AspectJ’s declare parents construct. This effectively forces the Patient class to implement the (empty) Subject interface. Similar mappings are defined for Nurse and Doctor. This allows methods defined in the abstract aspect, which are defined in terms of Subject and Observer objects, to work with Patient, Nurse, and Doctor objects.
The aspect next defines state components required to monitor multiple pattern instances. The first of these components is the `playersMap` which maps each object to the set of pattern instances it plays a role in, and the second component, the `patInstMap`, maintains a mapping from each lead object to its corresponding `PatternInstance` object. The inner `PatternInstance` class is shown next, which stores information about a single pattern instance. This information is used to evaluate expressions, such as those that make use of the `instance` keyword. The `playersMap` and `patInstMap` are updated accordingly when a pattern instance is created or destroyed, and when an object enrolls or disenrolls.

The aspect next declares the pointcuts corresponding to each of the role methods specified in the pattern contract. The advice bound to these pointcuts is responsible for checking the appropriate role method requirements, as well as for updating the pattern instance map and trace stack. Since the mapping between class methods and (non-constructor) role methods varies from application to application, the pointcuts are declared abstract. Pointcut definitions are supplied in the subaspect based on the interface maps specified in the relevant subcontract. The subaspect generated from the `Hospital` subcontract, for example, maps the `Subject.Attach()` pointcut (corresponding to `Subject.Attach()`) to the execution of either `Patient.addNurse()` or `Patient.addDoc()`. Pointcuts are also declared to capture the unnamed methods of the class(es) mapped to each role. These pointcuts are defined to include all of the class methods except those bound to role methods.
public abstract aspect Observer_Monitor {
    interface Subject{} interface Observer{}
    protected HashMap playersMap; // object -> Set of instance
    protected HashMap patInstMap; // lead -> instance

class PatternInstance {
    Set Subject, Observer;
    Subject lead;
    ...instance creation and maintenance methods provided...
}

// pointcuts for role constructors, role methods, and other methods:
abstract pointcut Subject_Attach_PC(Subject _this, Observer ob);
pointcut Subject_new_PC(Subject _this):
    execution(Subject+.new()) && target(_this);
...etc....

// auxiliary concept methods:
abstract boolean Modified(Subject arg1, Subject arg2);
abstract boolean Consistent(Subject arg1, Observer arg2);

// role state accessor methods:
abstract Set _observers(Subject _this, Subject lead);
abstract Subject _subjunct(Observer _this, Subject lead);

// assertion checking/bookkeeping advice code:
after(Observer _this): Observer_Update_PC(_this) {
    ...see Fig. 6.17 for details...
}
...etc. ...

// specialized TraceRecs:
class Subject_Attach_TraceRec extends TraceRec {
    ...similar to those in Chap. 3...
}
...etc. ...

Figure 6.16: The Observer_Monitor contract monitor (partial)
The requirements specified in the pattern contract are expressed in terms of auxiliary concepts and role fields; but since the realizations of these elements vary from application to application, they are captured using abstract methods, deferring their definitions to a subaspect. Observer_Monitor, for example, declares Modified() and Consistent() methods corresponding to the auxiliary concepts of the same name. It also declares abstract methods corresponding to Subject observers and Observer subject. Each of the latter methods returns the appropriate role field value when the argument passed as input is viewed as an instance of its role. As we have seen, the implementations of the auxiliary concept and role field methods are supplied in the subaspect based on the concept definitions and state maps provided in the relevant subcontract. Since these elements are defined (in the subcontract) in terms of code fragments, the code generation task is straightforward.

The next portion of the aspect defines the advice bound to each pointcut. The checking code within the advice is generated based on the assertions specified in the pattern contract. The before and after advice bound to each pointcut is responsible for checking the relevant pre- and postconditions, preserves clauses, and adjunct state updates. The advice is also responsible for calls that update playersMap, patInstMap, and TraceStack as needed. The aspect code for the after advice for Update() is given in Fig. 6.17, with typecasting code omitted. The advice bound to the remaining pointcuts is defined in a similar manner.

6.4 Code Size and Runtime Overhead

We have applied our approach to several different patterns and systems, and it is interesting to consider the relationship between contract/subcontract size and the size
after(Observer _this): Observer_Update_PC(.this) {
    Trace tau = TraceStackpop();
    Observer_Update_TraceRec caller = tau.caller;
    Observer _pre_this = caller._pre_this;

    PatternInstance instance; Subject lead;
    boolean pres_res, ens_res;

    Iterator it = playerMap.get(.this).iterator();
    while(it.hasNext()) {
        instance = it.next(); lead = instance.lead;

        // check preserves clause
        pres_res = _subject(_pre_this, lead) == (_subject(.this, lead));
        if(!pres_res) { ... print message... };

        // check postcondition
        ens_res = Consistent(_subject(_this, lead), _this);
        if(!ens_res) { ... print message... };
    }

    TraceStack.appendToTopTrace(caller);
    if (TraceStack.height==0) checkInv(.this);
}

Figure 6.17: after advice generated for Update() role method
of the corresponding monitoring code. It is also interesting to consider the runtime overhead introduced when this code is woven into an actual system. Figures 6.18 and 6.19 present the data corresponding to the use of our contracts for Observer, Memento, and Chain of Responsibility when used in monitoring the canonical system examples presented in [33]. As a gross estimate, we measure contract and aspect size in terms of the number of (non-comment) tokens. To arrive at the running times, we ran each application 10 times, and took the average. The tests were run on a Pentium-IV processor at 2.53GHz, with 512MB RAM, running Windows XP Pro SP 2, using the Sun Java Virtual Machine, version 1.5.0_04.
6.5 Pattern Checking with Other Monitoring Tools

6.5.1 Runtime Monitoring Using JML and jContractor

*jContractor* [46] is a toolkit for embedding assertion checking code within Java classes. Its purpose is to support the design-by-contract paradigm in a manner similar to other assertion checking toolkits. Unlike some other tools, however, jContractor operates at a lower level of abstraction. The assertions to be checked are expressed as *pure* Java code, rather than as mathematical expressions involving the types of language extensions provided by other tools. Hence, jContractor does not require a custom compiler to generate its assertion checking code. Instead, the checking code is implemented within the relevant classes, or within derived classes of those classes, in the form of method implementations that follow a prescribed naming scheme. The instrumentation tools provided by jContractor are responsible for weaving this code into the appropriate points within the class.

The jContractor toolkit supports the checking of pre- and postconditions, and class invariants. To associate a precondition check with a method \( m() \), the designer must implement a boolean function named \( m\text{.Precondition}() \), with an argument list that matches the argument list of the original method. Postcondition checking is done in a similar way, except that if \( m() \) returns a value, there must be an additional argument to receive the return value. To associate a class invariant with a class \( C \), the designer must implement a boolean function named \( C\text{.Invariant}() \), which accepts no arguments. The jContractor toolkit also provides a simple quantification library that simplifies the expression of certain predicates.

When the appropriate pre- and postcondition, and invariant checking methods have been added to each class, the source code is compiled using a standard Java
compiler. The jContractor toolkit includes a custom class loader that applies bytecode transformations to each class when the class is loaded. These transformations ensure that the assertion checking code provided by the designer is invoked at the appropriate points. The transformations applied by the jContractor toolkit follow a simple set of rules. First, if an _Invariant() method is defined, the toolkit inserts calls to this method at the beginning and end of each public method (other than those associated with assertion checking). If _Invariant() returns false, it indicates that the class invariant has been violated, and the generated code will throw an unchecked exception. Similar rules define the instrumentation behavior for checking pre- and postconditions.

As we have seen, the postcondition of a method often references the pre-state value of the target object. To allow designers to check these conditions, the jContractor toolkit uses a special field within each class. When referenced from within a postcondition checking method, the OLD field references a copy of the target object’s pre-state value. This field might, for example, be used to assert that the target object was unchanged by a method invocation. Because of the manner in which jContractor functions, if a class includes a postcondition checking method that references the OLD field, the class must declare the field, and its type must match the containing class. When instrumenting a method with an associated postcondition check that references OLD, the jContractor toolkit will insert code at the beginning of the method body to clone the target object, and to assign the clone to the OLD field. Hence, the class must declare a suitable implementation of clone(). According to the jContractor documentation, it is possible to factor the pre- and postcondition, and invariant checking methods into a derived class, rather than including the methods as part of the class being checked but we were unable to properly evaluate this feature. There
appears to be an instrumentation problem associated with the use of the OLD field when implementing assertion checking methods in a derived class.

Next consider the Java Modeling Language (JML) [54]. JML is a specification language used to describe behavior of Java modules using class invariants, method pre- and postconditions, and other constructs. The standard JML distribution includes an assertion checking compiler, which takes the Java source code along with its JML specifications as input, and creates Java object files that behave in the same way as in the original system, except that it allows for the monitoring of its JML specifications during execution.

The JML specifications for a class are provided within JML annotations found throughout the class code. JML annotations are delimited by special comment markers /*@ @*/, or found on a single line beginning with //@; such a scheme allows for a regular Java compiler to process the normal class code while ignoring the JML specifications. In addition to specifying the usual pre- and postconditions for methods, and class invariants, JML allows for the declaration of ghost and model variables that are only used to aid in specification and monitoring, and are not part of the regular system. Model variables act as abstract variables, where a representation mapping must be provided from the concrete state to the model variable value. Ghost variables play a similar role, but instead of using a mapping to provide a definition, they must be manually updated at places in the code selected by the person specifying the class. This ghost update code also must be contained within annotations, since they are not part of the regular system. JML allows us to define classes and interfaces, method bodies, and even code snippets within methods, within the annotations. When compiled with the assertion checking compiler, these type definitions as well as code
found within these annotations become part of the resulting object code, along with
the necessary assertion checking code for the pre- and postconditions and invariants.

Let us now turn to how we can use JML or jContractor for our purposes. First,
neither system provides facilities for recording method call trace information, nor for
expressing predicates over traces. As a result, we had to develop classes for recording
this information, which borrowed many of their components from the MonGen
implementation, and had to simplify some of the trace predicate expressions. More
seriously, since there is no notion of patterns in jContractor or JML, and thus no
notion of requirements associated with pattern usage, all of the requirements that we
want to monitor have to be expressed as part of the invariants of individual classes,
and the pre- and postconditions of methods of the classes. For the same reason, we
also have to include in the “specifications” associated with the individual classes, suit-
able code that would record information about the pattern instances and the objects
enrolled in them, as well as update the call traces, during execution. For example, in
our case study, at the beginning of the execution of individual methods of classes, not
only should the precondition of the method be checked, but we also must record the
trace information about this call. In jContractor, this is handled in the precondition
method; in JML, in an annotation at the beginning of the method body. Similarly,
at the end of execution of individual methods, we must not only check the postcondi-
tion, but also suitably update the trace information, as well as the information about
pattern instances. Again, jContractor handles this inside postcondition methods, and
JML in annotations at the end of the method body.
6.5.2 Performance Comparison and Discussion

For large systems, performance of runtime monitoring can become an important issue. Therefore, we wanted to see how well the three approaches compared with respect to each other. The numbers in this section should be considered somewhat preliminary, however. Although we have developed precise specifications for a number of standard patterns, in our tool comparison, we have only looked at one pattern, the Observer pattern, and have only considered one application, the TCL system. TCL is a variation of a canonical example of Observer found in the literature [33], where digital and analog clock objects are observing a timer.

We ran the TCL system with no monitoring, and with each of the three approaches to monitoring, using the same computer configuration as before. In each case, we ran the system for 10 runs each. Each run consisted of 50 pattern instances, where each subject object (a Time object) was attached to 40 observers (20 Clocks and 20 LazyPersons). The table in Fig. 6.20 summarizes the running times. We conjecture that the JML approach has considerably poorer performance because the JML compiler apparently inserts a significant amount of code into classes to facilitate checking that must be executed, increasing the runtime cost. Another is that JML is more

<table>
<thead>
<tr>
<th></th>
<th>No Monitoring</th>
<th>MonGen</th>
<th>JML</th>
<th>jContractor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Running Time</td>
<td>239.0</td>
<td>498.3</td>
<td>2290.7</td>
<td>433.0</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>14.8</td>
<td>91.3</td>
<td>373.4</td>
<td>129.7</td>
</tr>
</tbody>
</table>

Figure 6.20: Performance comparison data

189
conservative and checks for more potential problems than the ones we were explicitly monitoring.

Although we ran performance trials of the three approaches, we believe that performance should not be the deciding factor. The point is that the monitoring approach is intended to be used during testing and debugging of the system. Thus while poor performance beyond a certain point will not be acceptable, minor improvements in performance are not of particular interest. What is more interesting is how easy each approach is to use in practice. With respect to this metric, the MonGen approach seems to be the best of the three when monitoring pattern-based behaviors. The pattern contracts are, in a real sense, nothing more than formal versions of the standard informal descriptions that one finds in pattern catalogs. Furthermore, the pattern contracts do not have to be rewritten for the different applications that implement the pattern. Thus, given the availability of pattern contracts in, say, a publicly accessible online catalog (one of the future directions of this work), application developers would not have to worry about writing the contracts themselves. They would still be responsible for writing the subcontracts, but they simply state how the patterns are applied in a given system, and are relatively easy to write.

6.6 Related Work

Runtime assertion monitoring of OO systems has a long history [60, 70, 17], and some authors have considered aspect-based approaches. Lippert and Lopes [55] use
AspectJ to refactor pre- and postconditional assertion checking code. Gibbs and Malloy [35] propose using aspects to monitor class invariants involving temporal properties. To our knowledge, however, we are the first to investigate contract monitors for design patterns.
CHAPTER 7

SUMMARY AND FUTURE WORK

We will now conclude by giving a brief summary of the main contributions of this dissertation, with pointers to future work.

7.1 Specifying and Monitoring Polymorphic Behavior

In Chapters 2 and 3, we examined issues relating to the specification and monitoring of OO framework components. Frameworks provide general behaviors that can be useful across a variety of applications. These behaviors can often be characterized by the patterns of hook method calls made by the template methods of the framework, and are precisely captured in interaction specifications. Because such components are often provided without source code, monitoring such specifications would seem difficult, if not impossible. However, we have shown how to do so by exploiting polymorphism, and without having to rely on inserting special monitoring code into the component, or circumventing the language or runtime system entirely.

In Chapter 3, we presented the TCGen tool which generates classes that facilitate such monitoring, though, there are several important respects in which the tool can be improved. One possible avenue would be to investigate appropriate metrics against
which we can evaluate sets of test cases, and possibly have the tool automatically construct reasonable test cases. It would be preferable if these could be done effectively with minimal user input. Typical coverage criteria [57, 1, 71] for testing polymorphic code have been concerned with measuring the extent to which, for example, every hook method call is dispatched, in some test run, to each definition of the hook method. However, such a criterion would be inappropriate for us since our goal is to test the framework methods independently of any derived classes. A more appropriate approach would be to try and cover as many of the legal sequences of hook method calls in the test runs. One approach, often used with specification-based testing, is based on partitioning of the input space. It may be useful to investigate whether there is an analogous approach for testing against interaction specifications, though partition-based testing is known to suffer from some important problems [25, 37]. Another possibility would be to use a methodology similar to that employed in testing graphical user interfaces, where coverage based on event sequences is used for intra- and inter-component testing [59].

Other improvements would include making the tool produce testing classes that are more compatible with the JUnit [6] testing framework, which has become one of the most popular open-source software tools used by practicing developers. Another improvement would be to integrate regression testing into our methodology. This would seem to be a natural extension of the tool, considering we can empirically determine method dependencies relative to given test cases. Also, the trace-based assertion language can be difficult to use in practice in cases where there are many conditions placed on the possible sequences of hook calls. Special notations, such as regular expressions and statecharts, would be easier to understand, although using
them exclusively would likely come at the cost of expressibility. A more reasonable approach would be to allow for both types of specification to be used. Upon deciding what enhancements could be made to the specification language, the tool could be modified to work with these special notations.

### 7.2 Towards a Pattern-Centric Software Lifecycle

The goal of our work presented in Chapters 5 and 6 has been to improve the reliability of systems developed using design patterns. We first described a pattern contract language, PCL, that allows us to formalize patterns and the ways in which patterns are used in particular systems. Our approach allows us to characterize patterns precisely without compromising flexibility. Our second contribution was MonGen, a tool that given the pattern contracts and subcontracts used in designing a particular system, produces a set of aspects that can monitor the system at runtime to determine whether the pattern contracts are respected.

The contracts and subcontracts used in designing a system are valuable across the software lifecycle. Members of a maintenance team, for example, can use them to gain a more complete understanding of a system’s design. When a system is modified during maintenance and evolution, the MonGen tool can be mechanically used to generate the monitoring code for the new system. Indeed, if the modifications do not affect the ways in which patterns have been specialized, the aspects produced by MonGen based on the original contracts and subcontracts may be used without change. More importantly, the generated aspects allow the system maintainers to ensure that design integrity of the system is preserved.
These contributions, along with our planned extensions, provide the basis for a pattern-centric software lifecycle. At its foundation is a contract catalog that complements the existing pattern catalogs. The catalog is an evolving document that we plan to make accessible through the web. We hope that researchers interested in lightweight formal methods will contribute to its development, as well as software practitioners working in the field. Community involvement is essential in ensuring that the contracts faithfully capture the intent of the patterns specified. Members of a design team will be able to consult the catalog to ensure a common understanding of the requirements associated with the patterns underlying a particular design.

As the design and implementation details of a system are fleshed out, part of the design team will be charged with creating the corresponding subcontracts. In addition to guiding the implementation, the subcontracts will allow implementation and maintenance teams to generate appropriate runtime monitoring code. Executing this code will enable the team to identify pattern implementation errors more easily—from early implementation through later evolution.

Note that while developing a pattern contract requires reasonable facility with formal notations, developing a subcontract is a task that will be more appealing to system developers. Indeed, this is one advantage in having this portion of the formalism resembling a programming notation more than it resembles formal mathematics. As part of our future work, we plan to assess the degree of effort involved in developing and maintaining these subcontracts. This will allow us to perform a cost-benefit analysis by comparing this effort to the benefit received when using the approach. We also plan to investigate techniques for generating test suites that ensure suitable coverage of the patterns used in a system.
Another interesting possibility is the creation of a pattern-centric visualization tool. During a system’s execution, the monitoring aspects save information relevant to the patterns used in the system in the trace variables. With only slight modifications to the trace stack, all of this information could be collected (recall that in the current implementation, traces are discarded as they are popped off of the stack). The visualization tool could then take this information and “play” it in the form of a “video” that graphically represents the execution of the system. Events to be visualized would include pattern (de)instantiation and object (dis)enrollment, as well as object interactions via method calls. The tool could allow the user to fast-forward or rewind to certain points in the system’s execution, and allow them to focus in on the interactions among groups of objects interacting according to the patterns of interest. This will be of particular value to new members of a design team since it will enable them to quickly develop a pattern-centric understanding of relevant systems. Coupled with the contract checking features of MonGen, the tool could also show at what points a particular implementation violated the contract, which could prove very useful in debugging.
BIBLIOGRAPHY


