LNCS 2931

Alexandre Petrenko
Andreas Ulrich (Eds.)

Formal Approaches to Software Testing

Third International Workshop on
Formal Approaches to Testing of Software, FATES 2003
Montreal, Quebec, Canada, October 2003, Revised Papers

Springer

TP311.5-53
F723
2003

Alexandre Petrenko   Andreas Ulrich (Eds.)

Formal Approaches to Software Testing

Third International Workshop on
Formal Approaches to Testing of Software, FATES 2003
Montreal, Quebec, Canada, October 6th, 2003
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Alexandre Petrenko
Centre de recherche informatique de Montréal (CRIM)
550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada
E-mail: Petrenko@crim.ca

Andreas Ulrich
Siemens AG, Corporate Technology CT SE 1
Otto-Hahn-Ring 6, 81730 Munich, Germany
E-mail: andreas.ulrich@siemens.com

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.2.5, D.2.4, D.2, D.3.1, F.3.1

ISSN 0302-9743
ISBN 3-540-20894-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10984468 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 2931
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Preface

Formal methods provide system designers with the possibility to analyze system models and reason about them with mathematical precision and rigor. The use of formal methods is not restricted to the early development phases of a system, though. The different testing phases can also benefit from them to ease the production and application of effective and efficient tests. Many still regard formal methods and testing as an odd combination. Formal methods traditionally aim at verifying and proving correctness (a typical academic activity), while testing shows only the presence of errors (this is what practitioners do). Nonetheless, there is an increasing interest in the use of formal methods in software testing. It is expected that formal approaches are about to make a major impact on emerging testing technologies and practices. Testing proves to be a good starting point for introducing formal methods in the software development process.

This volume contains the papers presented at the 3rd Workshop on Formal Approaches to Testing of Software, FATES 2003, that was in affiliation with the IEEE/ACM Conference on Automated Software Engineering (ASE 2003). This year, FATES received 43 submissions. Each submission was reviewed by at least three independent reviewers from the program committee with the help of additional reviewers. Based on their evaluations, 18 papers submitted by authors from 13 different countries were selected for presentation at the workshop. The papers present different approaches to using formal methods in software testing. One of the main themes is the generation of an efficient and effective set of test cases from a formal description. Different models and formalisms are used, such as finite state machines, input/output transition systems, timed automata, UML, and Abstract State Machines. An increasing number of test methodologies (re)uses techniques from model checking. The prospects for using formal methods to improve software quality and reduce the cost of software testing are encouraging. But more efforts are needed, both in developing new theories and making existing methods applicable to the current practice of software development projects. Without doubt, coming FATES workshops will continue to contribute to the growing and evolving research activities in this field.

We wish to express our gratitude to the authors for their valuable contributions. We thank the program committee and the additional reviewers for their support in the paper selection process. Last but not least, we thank May Haydar who helped in organizing the proceedings and all persons from the Centre de Recherche Informatique de Montréal and the organizing committee of ASE 2003 who were involved in arranging local matters.

Montréal and München
October 2003

Alexandre Petrenko, Andreas Ulrich
FATES 2003 Co-chairs

Program Committee

Marie-Claude Gaudel	Université Paris-Sud, France
Jens Grabowski	Georg-August-Universität Göttingen, Germany
Dick Hamlet	Portland State University, USA
Robert M. Hierons	Brunel University, UK
Thierry Jéron	IRISA, France
David Lee	Bell Labs Research, China
Brian Nielsen	Aalborg University, Denmark
Jeff Offutt	George Mason University, USA
Alexandre Petrenko	CRIM, Canada
Jan Tretmans	Universiteit Nijmegen, The Netherlands
Andreas Ulrich	Siemens AG, Germany
Antti Valmari	Tampereen Teknillinen Yliopisto, Finland
Carsten Weise	Ericsson, Germany
Martin Woodward	Liverpool University, UK

Additional Reviewers

Serge Boroday
Rene de Vries
Lars Frantzen
Hesham Hallal
May Haydar
Ruibing Hao
Jia Le Huo
Claude Jard
Ahmed Khoumsi
Keqin Li
Andrej Pietschker
Vlad Rusu
Valery Tschaen
Machiel van der Bijl
Dong Wang
Gavin Yang
Xia Yin

Table of Contents

Program Testing and Analysis

Black-Box Testing of Grey-Box Behavior 1
Benjamin Tyler and Neelam Soundarajan

On Checking Whether a Predicate Definitely Holds 15
Alper Sen and Vijay K. Garg

Using a Software Testing Technique to Improve Theorem Proving 30
Reiner Hähnle and Angela Wallenburg

Auto-generating Test Sequences Using Model Checkers: A Case Study 42
Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao

Mutually Enhancing Test Generation and Specification Inference 60
Tao Xie and David Notkin

JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit 70
Guoqing Xu and Zongyuang Yang

Test Theory and Test Derivation Algorithms

Compositional Testing with IOCO 86
Machiel van der Bijl, Arend Rensink, and Jan Tretmans

Defining Observation Objectives for Reactive and Distributed Systems 101
Timo Aaltonen

Time-Optimal Real-Time Test Case Generation Using UPPAAL 114
Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou

Test Cases Generation for Nondeterministic Real-Time Systems 131
Ahmed Khoumsi, Thierry Jéron, and Hervé Marchand

Property Oriented Test Case Generation 147
Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon

Computing Unique Input/Output Sequences Using Genetic Algorithms ... 164
Qiang Guo, Robert M. Hierons, Mark Harman, and Karnig Derderian

Automatic Generation of Test Purposes for Testing Distributed Systems .. 178
Olaf Henniger, Miao Lu, and Hasan Ural

Test Methods and Test Tools

Interaction Testing in an Embedded System
Using Hardware Fault Injection and Program Mutation 192
Ahyoung Sung and Byoungju Choi

Automatic Conformance Testing of Internet Applications 205
Harm M.A. van Beek and Sjouke Mauw

A Use Case Driven Testing Process: Towards a Formal Approach
Based on UML Collaboration Diagrams 223
Mourad Badri, Linda Badri, and Marius Naha

VISWAS and on Diagnosability with IEEE Std P1522
and UML2.0 Testing Profile 236
Sita Ramakrishnan

Towards a Tool Environment for Model-Based Testing with AsmL 252
Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson,
Wolfram Schulte, Nikolai Tillmann, and Margus Veanes

Author Index 267

Black-Box Testing of Grey-Box Behavior

Benjamin Tyler and Neelam Soundarajan

Computer and Information Science
Ohio State University, Columbus, OH 43210, USA
{tyler,neelam}@cis.ohio-state.edu

Abstract. *Object-oriented frameworks* are designed to provide functionality common to a variety of applications. Developers use these frameworks in building their own specialized applications, often without having the source code of the original framework. Unfortunately, the interactions between the framework components and the new application code can lead to behaviors that could not be predicted even if valid *black-box* specifications were provided for the framework components. What is needed are *grey-box* specifications that include information about sequences of method calls made by the original framework code. Our focus is on how to test frameworks against such specifications, which requires the ability to monitor such method calls made by the framework during testing. The problem is that without the source code of the framework, we cannot resort to code instrumentation to track these calls. We develop an approach that allows us to do this, and demonstrate it on a simple case study.

1 Introduction

An important feature of object-oriented (OO) languages is the possibility of enriching or extending the functionality of an OO system [18] by providing, in derived classes, suitable definitions or re-definitions for some of the methods of some of the classes of the given system. *Application frameworks* [9, 13, 20] provide compelling examples of such enrichment. The framework includes a number of *hooks*, methods that are not (necessarily) defined in the framework but are invoked in specific, and often fairly involved, patterns by the *polymorphic* or *template* methods [11] defined in the framework. An application developer can build a complete customized application by simply providing appropriate (re-)definitions for the hook methods, suited to the needs of the particular application. The calls to the hook methods from the template methods are *dispatched* to the methods defined by the application developer, so that the template methods also exhibit behavior tailored to the particular application. Since the patterns of hook method calls implemented in the template methods are often among the most intricate part of the overall application, a well designed framework can be of great help in building applications, and maximizes the amount of reuse among the applications built on it. Our goal is to investigate approaches to perform specification-based testing of such frameworks.

Testing such systems should clearly include testing these patterns of hook method calls. That is, we are interested in testing what is called the *grey-box* behavior [2, 5, 10, 22] of OO systems, not just their *black-box* behavior. If we had access to the *source code* of the template methods, we could do this by instrumenting that code by inserting suitable instructions at appropriate points to record information about the hook method calls; for example, just prior to each such call, we could record the identity of the method being called, the values of the arguments, etc. But framework vendors, because of proprietary considerations, often will not provide the source code of their systems. Hence the challenge we face is to find a way to test the grey-box behavior of template methods without being able to make any changes to its code such as adding "monitoring code", indeed without even having the file containing source code of the system.

In this paper, we develop an approach that allows us to do this. The key idea underlying our approach is to exploit *polymorphism* to *intercept* hook method calls made by the template method being tested. When the hook method call is intercepted, the testing system will record the necessary information about the call, and then allow "normal" execution to resume. In a sense, the testing system that we build for testing a given framework *is itself an application built on the framework being tested.* This "application" can be generated automatically given information about the structure of the various classes that are part of the framework including the names and parameter types of the various methods and their specifications, and the *compiled* code of the framework. We have implemented a prototype *test system generator* that accomplishes this task. We present some details about our prototype later in the paper.

1.1 Black-Box vs. Grey-Box Behavior

How do we specify grey-box behavior? Standard specifications [14, 18] in terms of *pre-* and *post-conditions* for each method of each class in the system only specify the black-box behavior of the method in question. Consider a template (or polymorphic, we will use the terms interchangeably) method t(). There is no information in the standard specification of t() about the hook method calls that t() makes during execution. We can add such information by introducing a *trace* variable [5, 22], call it τ, as an *auxiliary* variable [19] on which we record information about the hook method calls t() makes. When the method starts execution, τ 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 τ. We can then specify the grey-box behavior by including, in the post-condition of t(), not just information on the state of the object in question when t() terminates, but also about the value of τ, i.e., about the hook method calls t() made during its execution; we will see examples of this later in the paper. Given such a grey-box specification, the key question we address is, how do we test t(), without accessing or modifying its code, to see if its actual grey-box behavior satisfies the specification?

1.2 Comparison to Related Work

A number of authors have addressed problems related to testing of polymorphic interactions [1, 21, 3, 17] in OO systems. In all of this work, the approach is to try to test the behavior of a polymorphic method t() by using objects of all or many different derived classes to check whether t() behaves appropriately in each case, given the different hook method definitions to which the calls in t() will be dispatched, depending on the particular derived class that the given object is an instance of. Such an approach is not suitable for testing frameworks. We are interested in testing the framework independently of any application that may be built on it, i.e., independently of particular derived classes and particular definitions of the hook methods. The only suitable way to do this is to test it directly to see that the actual sequences of hook method calls it makes during the tests are consistent with its grey-box specification. The other key difference is our focus on testing polymorphic methods without having access to their source code.

Another important question, of course, has to do with *coverage*. Typical coverage criteria that have been proposed [1, 21, 6] for testing polymorphic code have been concerned with measuring the extent to which, for example, every hook method call that appears in the polymorphic method is dispatched, in some test run, to each definition of the hook method (in the various derived classes). Clearly a criterion of this kind would be inappropriate for our purposes since our goal is to test the polymorphic methods of the framework independently of any derived classes. What we should aim for instead is to select test cases in such a way as to ensure that as many as possible of the sequences of hook method calls allowed by the grey-box specifications actually appear in the test runs. One problem here, as in any specification-based testing approach, is that the specification only specifies what behavior is *allowed*; there is no requirement that the system actually exhibit each behavior allowed by the specification. Hence, measuring our coverage by checking the extent to which the different sequences of hook method calls allowed by the specification show up in the test runs may be too conservative if the framework is not actually capable of exhibiting some of those sequences. Another approach, often used with specification-based testing, is based on partitioning of the input space, i.e., the set of values allowed by the pre-condition of the method. But partition-based testing suffers from some important problems [8, 12] that raise concerns about its usefulness. We will return to this question briefly in the final section but we should note that our focus in this paper is developing an approach that, without needing us to access or modifying the source code of a template method, allows us to check whether the method meets its grey-box specification during a test run, rather than coverage criteria.

1.3 Contributions

The main contributions of the paper may be summarized as follows:

- It identifies the importance of testing grey-box behavior of OO systems.
- It develops an approach to testing a system to see if it meets its grey-box specification without accessing or modifying the code of the system under test.
- It illustrates the approach by applying it to a simple case study.

In Sect. 2 we consider how to specify grey-box behavior. In Sect. 3, we develop our approach to testing against such specifications without accessing the code. We use a simple case study as a running example in Sects. 2 and 3. In Sect. 4 we present some details of our prototype system. In Sect. 5, we summarize our approach and consider future work.

2 Grey-Box Specifications

2.1 Limitations of Black-Box Specifications

Consider the Eater class, a simple class whose instances represent entities that lead sedentary lives consisting of eating donuts and burgers, depicted in Fig. 1. The methods Eat_Donuts() and Eat_Burgers() simply update the single member variable cals_Eaten which keeps track of how many calories have been consumed; the parameter n indicates how many donuts or burgers is to be consumed. Pig_Out() is a template method and invokes the hook methods Eat_Donuts() and Eat_Burgers().

```
class Eater {
  protected int cals_Eaten = 0;
  public void Eat_Donuts(int n) {
    cals_Eaten = cals_Eaten + 200 * n;}
  public void Eat_Burgers(int n) {
    cals_Eaten = cals_Eaten + 400 * n;}
  public final void Pig_Out() {
    Eat_Donuts(2); Eat_Burgers(2); }
}
```

Fig. 1. Base class Eater.

Let us now consider the specification of Eater's methods (Fig. 2). These can be specified as usual in terms of pre- and post-conditions describing the effect of each method on the member variables of the class. Here, we use the prime (′) notation in the post-conditions to refer to the value of the variable in question at the time the method was invoked. Thus the specifications of Eat_Donuts() and Eat_Burgers() state that each of them increments the value of cals_Eaten appropriately. Given the behaviors of these methods, it is easy to see that the template method Pig_Out() will meet its specification that it increments cals_Eaten by 1200.

Now suppose that the implementers of Eater provide only the compiled binary file and the black-box specification shown in Fig. 2, but not the source code in Fig. 1, to developers who wish to incorporate Eater in their own systems. What can such developers safely say about their own new classes that are extensions of Eater? Let us examine this question using the Eater_Jogger class, depicted in Fig. 3. Eater_Jogger, which is a derived class of Eater, keeps track not only of

$$\begin{aligned} \mathsf{pre.Eat\_Donuts(n)} &\equiv \mathsf{n} > 0 \\ \mathsf{post.Eat\_Donuts(n)} &\equiv \mathsf{cals\_Eaten} = \mathsf{cals\_Eaten}' + 200 * \mathsf{n} \end{aligned} \tag{2.1}$$

$$\begin{aligned} \mathsf{pre.Eat\_Burgers(n)} &\equiv \mathsf{n} > 0 \\ \mathsf{post.Eat\_Burgers(n)} &\equiv \mathsf{cals\_Eaten} = \mathsf{cals\_Eaten}' + 400 * \mathsf{n} \end{aligned} \tag{2.2}$$

$$\begin{aligned} \mathsf{pre.Pig\_Out()} &\equiv \mathit{true} \\ \mathsf{post.Pig\_Out()} &\equiv \mathsf{cals\_Eaten} = \mathsf{cals\_Eaten}' + 1200 \end{aligned} \tag{2.3}$$

Fig. 2. Eater's black-box specification.

```
class Eater_Jogger extends Eater {
  protected int cals_Burned = 0;
  public void Jog() {
    cals_Burned = cals_Burned + 500; }
  public void Eat_Donuts(int n) {
    cals_Eaten = cals_Eaten + 200 * n;
    cals_Burned = cals_Burned + 5 * n;}
  public void Eat_Burgers(int n) {
    cals_Eaten = cals_Eaten + 400 * n;
    cals_Burned = cals_Burned + 15 * n;}
}
```

Fig. 3. The derived class Eater_Jogger.

cals_Eaten but also the new data member cals_Burned. The new method Jog() simply increments cals_Burned. More important, Eat_Donuts() and Eat_Burgers() have been redefined to update cals_Burned.

What can we say about the behavior of Pig_Out() in this derived class? More precisely the question is, if ej is an object of type Eater_Jogger, what effect will the call ej.Pig_Out() have on ej.cals_Eaten and ej.cals_Burned? The calls in Pig_Out() to the hook methods will be dispatched to the methods redefined in Eater_Jogger. If we had access to the body of Pig_Out() (defined in the base class), we can see that it invokes Eat_Donuts(2) and then Eat_Burgers(2), and hence conclude, given the behaviors of these methods as redefined in Eater_Jogger, that in this class, Pig_Out() would increment cals_Eaten by 1200 and cals_Burned by 40. However, we have assumed that we only have access to Eater's black-box specification shown in Fig. 2, but *not* the source code of Pig_Out().

Behavioral subtyping [15] provides part of the answer to this question. In essence, a derived class D is a behavioral subtype of its base class B if every method redefined in D satisfies its B-specification. If this requirement is met then we can be sure that in the derived class, a template method t() will meet its original specification ((2.3) in the case of Pig_Out()). This is because when reasoning about the behavior of t() in the base class, we would have appealed to the base class specifications of the hook methods when considering the calls in t() to these methods. If these methods, as redefined in D, satisfy those specifications, then clearly that reasoning still applies when the calls that t() makes to these methods are dispatched to the redefined versions in D. Our redefined

Eat_Donuts() and Eat_Burgers() do clearly satisfy their base class specifications (2.1) and (2.2), hence Pig_Out() in the derived class will also meet its base class specification (2.3).

But this is only part of the answer. The redefined hook methods not only satisfy their base class specifications but exhibit *richer* behavior in terms of their effect on the new variable cals_Burned, which is easily specified (Fig. 4). Indeed, the whole point of redefining the hook methods was to achieve this richer behavior; after all, if all we cared about was the base class behavior, there would have been no need to redefine them at all. Not only is the hook methods' behavior enriched through their redefinition, but the behavior of the template method in the derived class will also be enriched even though its code was not changed. How then, can we reason about this richer behavior of the template method?

$$\begin{aligned} \mathsf{pre.Eat\_Donuts(n)} &\equiv \mathsf{n} > 0 \\ \mathsf{post.Eat\_Donuts(n)} &\equiv \mathsf{cals\_Eaten} = \mathsf{cals\_Eaten}' + 200 * \mathsf{n} \\ &\quad \wedge \mathsf{cals\_Burned} = \mathsf{cals\_Burned}' + 5 * \mathsf{n} \end{aligned} \tag{4.1}$$

$$\begin{aligned} \mathsf{pre.Eat\_Burgers(n)} &\equiv \mathsf{n} > 0 \\ \mathsf{post.Eat\_Burgers(n)} &\equiv \mathsf{cals\_Eaten} = \mathsf{cals\_Eaten}' + 400 * \mathsf{n} \\ &\quad \wedge \mathsf{cals\_Burned} = \mathsf{cals\_Burned}' + 15 * \mathsf{n} \end{aligned} \tag{4.2}$$

Fig. 4. Specifications for Eater_Jogger's hook methods.

If we examine the specifications for the redefined hook methods shown in Fig. 4, and (2.3), the black-box specification of Pig_Out(), can we arrive at the richer behavior of Pig_Out() in Eater_Jogger, in particular that it will increment cals_Burned by 40? The answer is clearly no, since there is nothing in (2.3) that tells us which, if any, hook methods Pig_Out() calls and how many times and with what argument values. Given (2.3), it is possible that it called Eat_Donuts() once with 6 as the argument and never called Eat_Burgers(); or Eat_Burgers() once with 3 as the argument, and Eat_Donuts() zero times; it is even possible that Pig_Out() didn't call either hook method even once and instead directly incremented cals_Eaten by 1200. Even an implementation that called Eat_Donuts() ten times with 2 as the argument each time and then decremented cals_Eaten by 2800 would work. All of these and more are possible, and depending on which of these Pig_Out() actually does, its effect on cals_Burned will be different. Note that for all of these cases, the original behavior (2.3) is still satisfied. *That* is ensured by behavioral subtyping. But if we are to arrive at the richer behavior of Pig_Out(), we need not just the black-box behavior of the template method in the base class as specified in (2.3), but also its grey-box behavior.

2.2 Reasoning with Grey-Box Specifications

Consider the *grey-box* specification (5.1) in Fig. 5. Here, τ is the *trace* of this template method. τ is the empty sequence, ε, when Pig_Out() begins execution. Each time Pig_Out() invokes a hook method, we add an element to record this hook method invocation. This element contains the name of the hook method called, the values of the member variables of the Eater class at the time of the call, their values at the time of the return from this call, the values of any additional arguments at the time of the call, their values at the time of the return, and the value of any additional result returned by the call. The grey-box post-condition gives us information about the value of τ when the method finishes, hence about the hook method calls it made during its execution. Thus (5.1) states that $|\tau|$, the length of, i.e. the number of elements in, τ is 2; that the hook method called in the first call, recorded in the first element $\tau[1]$ of the trace, is Eat_Donuts; that the argument value passed in this call is 2; the hook method called in the second call is Eat_Burgers; and the argument passed in this call is 2.

$$\begin{aligned}\textsf{pre.Pig\_Out()} &\equiv \tau = \varepsilon \\ \textsf{post.Pig\_Out()} &\equiv \ \textsf{cals\_Eaten} = \textsf{cals\_Eaten}' + 1200 \wedge |\tau| = 2 \\ &\quad \wedge \tau[1].method = \text{“}\textsf{Eat\_Donuts}\text{”} \wedge \tau[1].arg = 2 \\ &\quad \wedge \tau[2].method = \text{“}\textsf{Eat\_Burgers}\text{”} \wedge \tau[2].arg = 2\end{aligned} \tag{5.1}$$

Fig. 5. Grey-box specification for Pig_Out in Eater.

It should be noted that (5.1) does not give us additional information about the value that cals_Eaten had at the time of either call or return. While this simplifies the specification, it also means that redefinitions of the hook methods that depend on the value of cals_Eaten cannot be reasoned about given (5.1). This is a tradeoff that we have to make when writing grey-box specifications; include full information, resulting in a fairly complex specification; or leave out some of the information, foreclosing the possibility of some enrichments (or at least of reasoning about such enrichments, which amounts to the same thing in the absence of access to the source code of the template method).

Given this grey-box specification, what can we conclude about the behavior of Pig_Out() in the derived class? Note first that from (4.1) and (4.2), we can deduce that Eater_Jogger.Eat_Donuts() and Eater_Jogger.Eat_Burgers() satisfy (2.1) and (2.2), i.e., they satisfy the requirement of behavioral subtyping; hence Pig_Out() will satisfy (2.3) when invoked on Eater_Jogger objects. But we can also conclude given (2.1) and (2.2) and, as specified by (5.1), that Pig_Out() will make two hook method calls during its execution, first to Eat_Donuts() with argument value 2, and then to Eat_Burgers() with argument value 2, that in Eater_Jogger, Pig_Out() will increment cals_Burned by 40, as specified in Fig. 6.

In [22], we have proposed a set of rules that can be used in the usual fashion of axiomatic semantics to show: first, that the body of Pig_Out() defined in Fig. 1 satisfies the grey-box specification (5.1); and second, by using the *enrichment rule* to "plug-in" the richer behavior specified in (4.1) and (4.2) for the redefined hook methods into (5.1), that in the derived class, the template method will satisfy the richer specification (6.1). Here our goal is to test the template method to see whether it satisfies its specification, so we now turn to that.