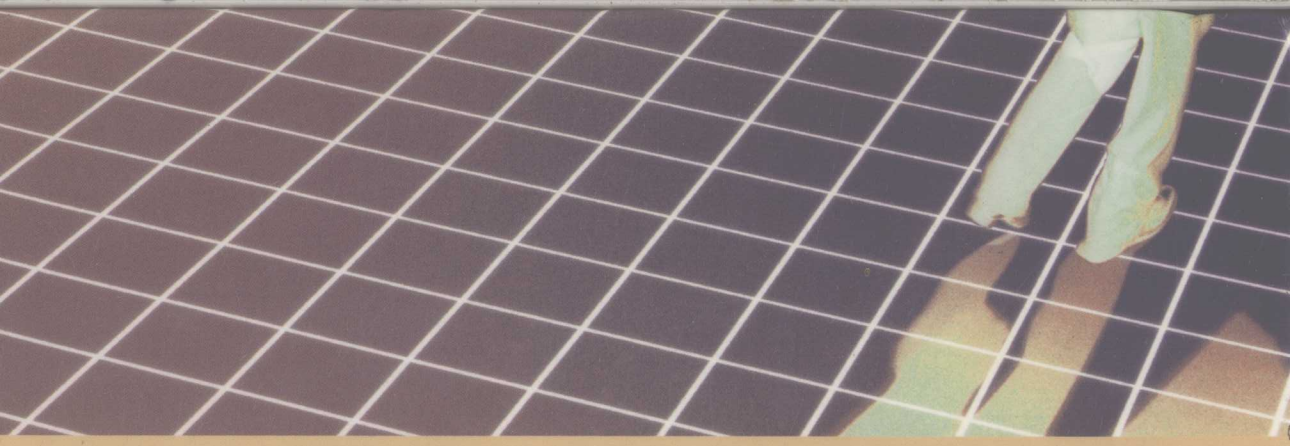

DESIGN PATTERNS FORMALIZATION TECHNIQUES

Toufik Taibi

Design Patterns Formalization Techniques

Toufik Taibi
United Arab Emirates University, UAE

IGI PUBLISHING
Hershey • New York

Acquisition Editor:	Kristin Klinger
Senior Managing Editor:	Jennifer Neidig
Managing Editor:	Sara Reed
Assistant Managing Editor:	Sharon Berger
Development Editor:	Kristin Roth
Copy Editor:	April Schmidt and Lanette Ehrhardt
Typesetter:	Jamie Snavely
Cover Design:	Lisa Tosheff
Printed at:	Yurchak Printing Inc.

Published in the United States of America by
IGI Publishing (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-pub.com
Web site: http://www.igi-pub.com

and in the United Kingdom by
IGI Publishing (an imprint of IGI Global)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 0609
Web site: http://www.eurospanonline.com

Library of Congress Cataloging-in-Publication Data

Design patterns formalization techniques / Toufik Taibi, editor.
p. cm.
Summary: "This book focuses on formalizing the solution element of patterns, providing tangible benefits to pattern users, researchers, scholars, academicians, practitioners and students working in the field of Design patterns and software reuse; it explains details on several specification languages, allowing readers to choose the most suitable formal technique to solve their specific inquiries"—Provided by publisher.
Includes bibliographical references and index.
ISBN 978-1-59904-219-0 (hardcover)—ISBN 978-1-59904-221-3 (ebook)
1. Software patterns. 2. Formal methods (Computer science) 3. Software architecture. I. Taibi, Toufik, 1969-
QA76.76.P37D47 2007
005.1—dc22

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Foreword

Software design is a fledgling discipline. When the "software crisis" came to be acknowledged during the late 1960s, software development projects have been marred by budget overflows and catastrophic failures. This situation has largely remained unchanged. Programmers still create poorly-understood systems of monstrous complexity which suffer from a range of problems directly linked to the lack of abstraction: lack of means of communicating design decisions, absence of effective pedagogic tools for training novice programmers, and inadequate means for maintaining gigantic software systems. In the recent years, we have witnessed an explosion of loosely-related software technologies, techniques, notations, paradigms, idioms, methodologies, and most of all proprietary and poorly-understood ad-hoc solutions, driven by market forces more than by design, planning, or research.

Design patterns were introduced to programming practices at the end of the 1980s as a result of dissatisfaction with software's state of affairs. The few means of abstraction in existence at the time, such as algorithms and data structures, narrowly-suited procedural programming, poorly fitting with the growing use of object-oriented programming paradigm. For the first time, an abstraction technique at hand was general enough to be useful for practitioners and academics alike, specific enough to enter textbooks, broad enough to be useful during any stage in the development process, and generic enough to support any programming paradigm. The introduction of Design patterns marks a turning point in the history of software design.

In 1995, we witnessed the publication of a catalogue (Gamma, Helm, Johnson, & Vlissides, 1995) of 23 Design patterns written by four experienced object-oriented designers. The catalogue, which came to be known as the "Gang of Four" catalogue, was an immediate success. The abstractions described offer a rich vocabulary for abstractions for conceptualising, designing, brain-storming, communicating, documenting, understanding, maintaining, and teaching about software. Each pattern captures a design motif that is common enough to deserve wide recognition, described in clarity and sifficient detail to indicate the consequences of choosing to apply it. Patterns help novices avoid common pitfalls and encourage experienced programmers to build better software. As a result, Design patterns entered

textbooks and became the subject matter of scientific papers, conferences, and intensive efforts for providing tool-support by a broad range of industrial software development environments. In the decade since they have entered the zeitgeist, Design patterns have revolutionized software design.

Early on, the attempt to reason about and provide tool support for Design patterns have led many to recognize that verbal descriptions and case studies are not enough. The software engineering community came to realize that conceptual clarity and automation require a formal specification language. The central problem in software design has therefore shifted from seeking suitable abstractions to providing precise means for capturing and representing them. But existing modelling notations which were designed for documenting design decisions tailored for specific programs, proved inadequate for the purpose of modelling abstract design motifs. This shortcoming has motivated the investigation in formal modelling techniques which is the subject matter of this book.

Mathematics is the most successful conceptual tool for capturing, representing, understanding, and using abstractions. Effective mathematical analysis and modelling is the hallmark of modern science and a mark of maturity of an engineering discipline. The research in formal techniques for modelling Design patterns is therefore the next natural step in the progress of software design. This line of investigation is vital for achieving conceptual clarity and ever more potent means of abstraction. This book provides a summary of this investigation.

A convergence of formalization techniques for Design patterns, expected to evolve within a decade or two, is vital for establishing a sound foundation for using and understanding patterns. Convergence is also crucial for communicating about and teaching patterns. Given the central role of Design patterns, such an achievement is widely taken to be the most important vehicle of progress for software design and a prerequisite for a regimented engineering discipline. The next generation of software design techniques may very well depend on accomplishing convergence. We hope this book shall speed and facilitate this process.

Amnon Eden

Layer-de-la-Haye, December 2006

***Amnon H. Eden** is a lecturer with the University of Essex and a research fellow with the Center For Inquiry. He is investigating the problem of modelling Design patterns since 1996. In 1998, Eden received his PhD from the Department of Computer Science, at Tel Aviv University, for his dissertation on language for patterns uniform specification (LePUS) and the building-blocks of object-oriented design. Eden's contributions also include the intension/locality hypothesis, the notion of evolution complexity, and the ontology of software. Eden lives with his partner in Layer-de-la-Haye, UK.*

Reference

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented systems*. Addison-Wesley Professional.

Preface

Christopher Alexander was the first to introduce patterns as a form of describing accumulated experiences in the field of architecture. He defines a *pattern* as a construct made of three parts: a *context*, a set of *forces* and a *solution* (Alexander, Ishikawa, & Silverstein, 1977). The *context* reflects the conditions under which the pattern holds (Alexander, 1979). The *forces* occur repeatedly in the context and represent the *problem(s)* faced (Alexander, 1979). The *solution* is a configuration that allows the forces to resolve themselves (i.e., balances the forces) (Alexander, 1979). Alexander's patterns comprise commonly encountered problems and their appropriate solutions for the making of successful towns and buildings in a western environment. Alexander called a set of correlated patterns a *pattern language*, because patterns form a vocabulary of concepts used in communications that take place between experts and novices.

Ward Cunningham and Kent Beck (1987) were inspired by Alexander's work and decided to adapt it to software development. Their first five patterns dealt with the design of user interfaces. This marked the birth of patterns in the software field. Nowadays, software patterns are so popular that they are being applied in virtually every aspect of computing. Moreover, the concept of patterns is being adapted to many other fields, such as management, education, and so forth.

There are many kinds of software patterns based on different categorizing criteria. If the focus is on the software design phase, patterns are classified according to their abstraction level into architectural patterns, Design patterns, and idioms. This book mainly focuses on Design patterns. As such, in this preface, the terms Design patterns and patterns can be used interchangeably.

A pattern can be defined as a description of a proven (successful or efficient) *solution* to a recurring *problem* within a *context*. The above definition keeps the essence of Alexander's original definition by mentioning the three pillars of a pattern (context, problem, and solution). Reusing patterns usually yields better quality software within a reduced time frame. As such, they are considered artifacts of software reusability.

Patterns are published mostly within collections or catalogs. However, the most influential publication of Design patterns is the catalog by the "*Gang of Four*" (GoF) (Gamma, Helm,

Johnson, & Vlissides, 1995), which listed 23 patterns classified under three categories: creational, structural, and behavioral. All patterns recorded in the GoF catalog were described in great detail and share an identical format of presentation.

Patterns are used as a way of improving software design productivity and quality for the following reasons:

- Patterns capture previous design experiences, and make it available to other designers. Therefore, designers do not need to discover solutions for every problem from scratch.
- Patterns form a more flexible foundation for reuse, as they can be reused in many ways.
- Patterns serve as a communication medium among software designers.
- Patterns can be considered microarchitectures, from which bigger software architectures can be built.

Well-established engineering disciplines have handbooks that describe successful solutions to known problems. Though as a discipline, software engineering is a long way from that goal, patterns have been useful for software engineers to reuse successful solutions.

Currently most patterns are described using a combination of textual descriptions, object-oriented (OO) graphical notations such as unified modeling language (UML) (Rumbaugh, Jacobson, & Booch, 1998), and sample code fragments. The intention is to make them easy to read and use and to build a pattern vocabulary. However, informal descriptions give rise to ambiguity, and limit tool support and correct usage. Tool support can play a great role in automated pattern mining, detection of pattern variants, and code generation from pattern specification.

Hence, there is a need for a formal means of accurately describing patterns in order to achieve the following goals:

- Better understand patterns and their composition. This will help know when and how to use them properly in order to take full advantage of their inherent benefits.
- Resolve the following issues regarding relationships between patterns such as duplication, refinement, and disjunction. Resolving the above-mentioned questions will ease the process of pattern repository management.
- Allow the development of tool support in activities related to patterns.

Many formal approaches for pattern specification have been emerging as a means to cope with the inherent shortcomings of informal descriptions. Despite being based on different mathematical formalisms, they share the same goal, which is accurately describing patterns in order to allow rigorous reasoning about them, their instances, their relationships and their composition and facilitate tool support for their usage. It is important to note that formal approaches to Design pattern specifications are not intended to replace existing informal approaches, but to complement them.

Currently, there is no single avenue for authors actively involved in the field of formal specification of Design patterns to publish their work. There has been neither a dedicated conference nor a special journal issue that covers precisely this field. Since this book contains chapters describing different Design pattern formalization techniques, it will contribute to the state-of-the-art in the field and will be a one-stop for academicians, research scholars, students, and practitioners to learn about the details of each of the techniques.

The book is organized into XVI chapters. A brief description of each of the chapters follows. These were mainly taken from the abstracts of the chapters.

Chapter I describes Balanced pattern specification language (BPSL), a language intended to accurately describe patterns in order to allow rigorous reasoning about them. BPSL incorporates the formal specification of both structural and behavioral aspects of patterns. The structural aspect formalization is based on first-order logic (FOL), while the behavioral aspect formalization is based on temporal logic of actions (TLA). Moreover, BPSL can formalize pattern composition and instances of patterns (possible implementations of a given pattern).

Chapter II describes the Design pattern modeling language (DPML), a notation supporting the specification of Design pattern solutions and their instantiation into UML design models. DPML uses a simple set of visual abstractions and readily lends itself to tool support. DPML Design pattern solution specifications are used to construct visual, formal specifications of Design patterns. DPML instantiation diagrams are used to link a Design pattern solution specification to instances of a UML model, indicating the roles played by different UML elements in the generic Design pattern solution. A prototype tool is described, together with an evaluation of the language and tool.

Chapter III shows how formal specifications of GoF patterns, based on the rigorous approach to industrial software engineering (RAISE) language, have been helpful to develop tool support. Thus, the object-oriented design process is extended by the inclusion of pattern-based modeling and verification steps. The latter involving checking design correctness and appropriate pattern application through the use of a supporting tool, called DePMoVe (design and pattern modeling and verification).

Chapter IV describes an abstraction mechanism for collective behavior in reactive distributed systems. The mechanism allows the expression of recurring patterns of object interactions in a parametric form, and to formally verify temporal safety properties induced by applications of the patterns. The authors present the abstraction mechanism and compare it to Design patterns, an established software engineering concept. While there are some obvious similarities, because the common theme is abstraction of object interactions, there are important differences as well. Authors discuss how the emphasis on full formality affects what can be expressed and achieved in terms of patterns of object interactions. The approach is illustrated with the OBSERVER and MEMENTO patterns.

In **Chapter V**, authors have investigated several approaches to the formal specification of Design patterns. In particular, they have separated the structural and behavioral aspects of Design patterns and proposed specification methods based on first-order logic, temporal logic, temporal logic of action, process calculus, and Prolog. They also explore verification techniques based on theorem proving. The main objective of this chapter is to describe their investigations on formal specification techniques for Design patterns, and then demonstrate using these specifications as the methods of reasoning about Design pattern properties when they are used in software systems.

Chapter VI presents the SPINE language as a way of representing Design patterns in a suitable manner for performing verification of a pattern's implementation in a particular source language. SPINE is used by a proof engine called HEDGEHOG, which is used to verify whether a pattern is correctly implemented.

Chapter VII presents a viewpoint based on intent-oriented design (IOD) that yields simple formalisms and a conceptual basis for tools supporting design and implementation from an intent-oriented perspective. The system for pattern query and recognition (SPQR) is an automated framework for analysis of software systems in the small or the large, and detection of instances of known programming concepts in a flexible, yet formal, manner. These concepts, when combined in well-defined ways to form abstractions, as found in the Design patterns literature, lead to the automated detection of Design patterns directly from source code and other design artifacts. The chapter describes the three major portions of SPQR briefly, and uses it to facilitate a discussion of the underlying formalizations of Design patterns with a concrete example, from source code to completed results.

Chapter VIII describes techniques for the verification of refactorings or transformations which introduce Design patterns. The techniques use a semantics of object-oriented systems defined by the object calculus and the pattern transformations are proved to be refinements using this semantics.

Chapter IX describes a UML-based pattern specification language called role-based metamodeling language (RBML), which defines the solution domain of Design patterns in terms of roles at the metamodel level. The chapter discusses benefits of the RBML and presents notation for capturing various perspectives of pattern properties. The OBSERVER, INTERPRETER, and ITERATOR patterns are used to describe RBML. Tool support for the RBML and the future trends in pattern specification are also discussed.

In **Chapter X**, the formal specification of a Design pattern is given as a class operator that transforms a design given as a set of classes into a new design that takes into account the description and properties of the Design pattern. The operator is specified in the SLAM-SL specification language, in terms of pre- and postconditions. Precondition collects properties required to apply the pattern and post-condition relates input classes and result classes encompassing most of the intent and consequences sections of the pattern.

Chapter XI describes a formal, logic-based language for representing pattern structure and an extension that can also represent other aspects of patterns, such as intent, applicability, and collaboration. This mathematical basis serves to eliminate ambiguities. The chapter explains the concepts underlying the languages and shows their utility by representing two classical patterns, some concurrent patterns and various aspects of a few other patterns.

Chapter XII introduces an approach to define Design patterns using Semantic Web technologies. For this purpose, a vocabulary based on the Web ontology language OWL is developed. Design patterns can be defined as RDF documents instantiating this vocabulary, and can be published as resources on standard Web servers. This facilitates the use of patterns as knowledge artefacts shared by the software engineering community. The instantiation of patterns in programs is discussed, and the design of a tool is presented that can x-ray programs for pattern instances based on their formal definitions.

Chapter XIII presents a novel approach allowing the precise specification of patterns as well as retaining the patterns' inherent flexibility. The chapter also discusses tools that can assist practitioners in determining whether the patterns used in designing their systems have been implemented correctly. Such tools are important also during system maintenance and

evolution to ensure that the design integrity of a system is not compromised. The authors also show how their approach lends itself to the construction of such tools.

Chapter XIV introduces the user requirements notation (URN), and demonstrates how it can be used to formalize patterns in a way that enables rigorous trade-off analysis while maintaining the genericity of the solution description. URN combines a graphical goal language, which can be used to capture forces and reason about trade-offs, and a graphical scenario language, which can be used to describe behavioral solutions in an abstract manner. Although each language can be used in isolation in pattern descriptions (and have been in the literature), the focus of this chapter is on their combined use. It includes examples of formalizing Design patterns with URN together with a process for trade-off analysis.

Chapter XV describes an extended compiler that formalizes patterns, called pattern enforcing compiler (PEC). Developers use standard Java syntax to mark their classes as implementations of particular Design patterns. The compiler is then able to use reflection to check whether the classes do in fact adhere to the constraints of the patterns. The checking possible with our compiler starts with the obvious static adherence to constraints, such as method presence, visibility, and naming. However, PEC supports dynamic testing to check the runtime behavior of classes and code generation to assist in the implementation of complex patterns. The chapter gives examples of using the patterns supplied with PEC, and also examples of how to write your own patterns and have PEC enforce these.

Chapter XVI presents Class-Z, a formal language for modelling OO Design patterns. The chapter demonstrates the language's unique efficacy in producing precise, concise, scalable, generic, and appropriately abstract specifications modelling the GoF Design patterns. Mathematical logic is used as a main frame of reference: the language is defined as a subset of first-order predicate calculus and implementations (programs) are modelled as finite structures in model theory.

References

Alexander, C. (1979). *The timeless way of building*. Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: Towns, buildings, construction*. Oxford University Press.

Beck, K., & Cunningham, W. (1987). Using pattern languages for object-oriented programs (Tech. Rep. No. CR-87-43). Tektronix Inc, Computer Research Laboratory.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented systems*. Addison-Wesley Professional.

Rumbaugh, J., Jacobson, I., & Booch, G. (1998). *The unified modeling language reference manual*. Addison-Wesley Professional.

Acknowledgment

The editor would like to acknowledge the help of all those who contributed to this one year long project. Without their help and support, the project could not have been satisfactorily completed.

Most of the authors of chapters included in this book also served as referees for articles written by other authors. Many thanks to all those who provided constructive and comprehensive reviews. Special thanks also go to all the staff at IGI Global, whose contributions throughout the whole process from inception of the initial idea to final publication have been very invaluable. In particular, to our development editor, Kristin Roth, whose continuous communication via e-mail kept the project on schedule.

I wish also to thank all of the authors for their insights and excellent contributions to this book.

Finally, I want to thank my wife and children for their love and support throughout this project.

Toufik Taibi, PhD

Al Ain, UAE

December 2006

Design Patterns Formalization Techniques

Table of Contents

Foreword vi

Preface viii

Chapter I
An Integrated Approach to Design Patterns Formalization 1
Toufik Taibi, United Arab Emirates University, UAE

Chapter II
A Visual Language for Design Pattern Modeling and Instantiation 20
David Maplesden, Orion Systems Ltd., New Zealand
John Hosking, University of Auckland, New Zealand
John Grundy, University of Auckland, New Zealand

Chapter III
A Generic Model of Object-Oriented Patterns Specified in RSL 44
Andrés Flores, University of Comahue, Argentina
Alejandra Cechich, University of Comahue, Argentina
Gabriela Aranda, University of Comahue, Argentina

Chapter IV
Patterns of Collective Behavior in Ocsid 73
Joni Helin, Tampere University of Technology, Finland
Pertti Kellomäki, Tampere University of Technology, Finland
Tommi Mikkonen, Tampere University of Technology, Finland

Chapter V
Formal Specification and Verification of Design Patterns 94
Jing Dong, University of Texas at Dallas, USA
Paulo Alencar, University of Waterloo, Canada
Donald Cowan, University of Waterloo, Canada

Chapter VI
SPINE: Language for Pattern Verification 109
Alex Blewitt, Edinburgh University, UK

Chapter VII
Intent-Oriented Design Pattern Formalization Using SPQR 123
Jason Smith, IBM T.J. Watson Research, USA
David Stotts, University of North Carolina at Chapel Hill, USA

Chapter VIII
Formalising Design Patterns as Model Transformations 156
Kevin Lano, King's College, UK

Chapter IX
The Role-Based Metamodeling Language for Specifying Design Patterns 183
Dae-Kyoo Kim, Oakland University, USA

Chapter X
Modeling and Reasoning about Design Patterns in SLAM-SL 206
Angel Herranz, Universidad Politécnica de Madrid, Spain
Juan José Moreno-Navarro, IMDEA Software, Spain

Chapter XI
The Applications and Enhancement of LePUS for Specifying Design Patterns 236
Rajeev R. Raje, Indiana University-Purdue University Indianapolis, USA
Sivakumar Chinnasamy, Verizon Information Services, USA
Andrew M. Olson, Indiana University-Purdue University Indianapolis, USA
William Hidgon, University of Indianapolis, USA

Chapter XII
An Ontology Based Representation of Software Design Patterns 258
Jens Dietrich, Massey University, New Zealand
Chris Elgar, SolNet Solutions Limited, New Zealand

Chapter XIII
Precision, Flexibility, and Tool Support: Essential Elements of Pattern Formalization 280
Neelam Soundarajan, The Ohio State University, USA
Jason O. Hallstrom, Clemson University, USA

Chapter XIV
Formalizing Patterns with the User Requirements Notation 302
Gunter Mussbacher, University of Ottawa, Canada
Daniel Amyot, University of Ottawa, Canada
Michael Weiss, Carleton University, Canada

Chapter XV
A Pattern Enforcing Compiler (PEC) for Java: A Practical Way to Formally Specify Patterns 324
Howard Lovatt, Macquarie University, Australia
Anthony M. Sloane, Macquarie University, Australia
Dominic R. Verity, Macquarie University, Australia

Chapter XVI
LePUS: A Formal Language for Modeling Design Patterns 357
Epameinondas Gasparis, University of Essex, UK

About the Authors 373

Index 381

Chapter I

An Integrated Approach to Design Patterns Formalization

Toufik Taibi, United Arab Emirates University, UAE

Abstract

A Design pattern describes a set of proven solutions for a set of recurring design problems that occur within a context. As such, reusing patterns improves both quality and time-to-market of software projects. Currently, most patterns are specified in an informal fashion, which gives rise to ambiguity, and limits tool support and correct usage. This chapter describes balanced pattern specification language (BPSL), a language intended to accurately describe patterns in order to allow rigorous reasoning about them. BPSL incorporates the formal specification of both structural and behavioral aspects of patterns. Moreover, it can formalize pattern composition and instances of patterns (possible implementations of a given pattern).

Introduction

A Design pattern describes a set of proven solutions for a set of recurring design problems that occurs within a certain context. Hence, reusing patterns yields better quality software within reduced time frames.

Currently, most patterns are described using a combination of textual descriptions, object-oriented (OO) graphical notations such as unified modeling language (UML) (Rumbaugh, Jacobson, & Booch, 1998), and sample code fragments. The intention was to make them easy to read and use, and to build a pattern vocabulary. However, informal descriptions give rise to ambiguity, and limit tool support and correct usage. Tool support can play a great role in automated pattern mining, detection of pattern variants, and code generation from pattern specification.

The pattern community mostly focuses on the solution element of a pattern and not on its other elements, such as the problem solved, the context, the important forces (Alexender, Ishikawa, & Silverstein, 1977) acting within the problem, or the way the pattern resolves these forces. Indeed, the verbal description of the solution element is the most coherent and the easiest to formalize. As such, this work also focuses on specifying the solution element of patterns.

Most formal approaches for pattern specification lack in the area of approachability due to the assumption that complex mathematical notations are necessary to achieve precision, thus favoring mathematically mature modelers rather than normal modelers (Taibi & Ngo, 2003b). Another problem of formal approaches is that they are not comprehensive enough to describe both aspects (structural and behavioral) of patterns. Additionally, only a few of the formal approaches attempted formalizing pattern composition (Taibi & Ngo, 2003b).

Balanced pattern specification language (BPSL) (Taibi & Ngo, 2003a) was developed in order to formally specify the structural as well as behavioral aspects of patterns at three levels of abstraction: pattern composition, patterns, and pattern instances.

First order logic (FOL) (Smullyan, 1995) is used as the formal basis for specifying the structural aspect of patterns, because relations between pattern participants can be easily expressed as predicates. Temporal logic of actions (TLA) (Lamport, 1994) is used as the formal basis for specifying the behavioral aspect of patterns, because it is best suited to describe the collective behavior of objects. BPSL has been successfully used to specify patterns for stand-alone systems (Taibi & Ngo, 2003a) and also for distributed object computing systems (Schmidt, Stal, Rohnert, & Buschmann, 2000; Taibi & Ngo, 2004).

The design of component-based software involves the composition of different components. Patterns are special types of components offering a flexible means of reuse. Since each pattern represents a well-tested abstraction that has many instances, patterns can be considered building blocks from which reusable and flexible software designs can be built. Checking the correctness of pattern composition allows detecting problems early in the lifecycle, which saves time and the cost of fixing errors at later stages. Thus, if formalized, pattern composition can lead to ready-made architectures from which only instantiation is required to build robust implementations. Since the specification of the structural and behavioral aspects of patterns uses two different formalisms (FOL and TLA), pattern composition is formalized independently for each aspect.

The rest of the chapter is organized as follows. Next, the chapter gives a detailed description of BPSL's concepts and constructs, while the following section describes BPSL's composition process. The chapter then provides case studies on applying BPSL for formally specifying patterns and their composition. This is followed with a description of how BPSL can be used to specify pattern instances and related work. Finally, the chapter concludes, providing future research directions.

Balanced Pattern Specification Language (BPSL)

Structural Aspect Formalization

The structural aspect of patterns is specified using a first-order language called S_{BPSL}, where "S" stands for "structural." The following are the formation rules that define the syntax of formulas in S_{BPSL}:

- A *term* is either a *constant* like *2* or a *variable* like *x*.
- An *atom* is a dyadic (binary) *predicate* symbol applied to two arguments, each of which is a term.
- A *formula* is either an atom, $A \wedge B$ (A and B are formulas), or any formula A and any variable x in $\exists x\, A$.

As it can be seen from the above formation rules, S_{BPSL} is a very simple first-order language having the following characteristics:

- Each formula is a well-formed formula.
- Each formula is a sentence, as it does not contain free variables.
- S_{BPSL} does not support function symbols, restricts predicates to have two arguments and requires only the usage of the existential quantifier ($\exists$).

Variables and constants of S_{BPSL} are many-sorted. Variable and constant symbols represent classes, typed variables and methods. The sets of classes (or references to classes), typed variables and methods are designated *C, V,* and *M,* respectively. Typed variables represent variables of any predefined or user-defined types except elements of set *C*. Binary predicate symbols represent permanent relations among them. BPSL defines a set of *primary* permanent relations based on which other permanent relations can be built (Table 1). The term "permanent" is used to differentiate these relations with "temporal" relations, defined below.

In Table 1, $M \times C$ is the Cartesian product of M and C. Primary permanent relations represent the smallest set (in terms of cardinality), on top of which any other permanent relation can be built. For example, the permanent relation *Forwarding* is a special case of *Invocation*,