A Survey on Design Pattern Formalisms

A.V.Sriharsha, Dr. A.Rama Mohan Reddy

Abstract— In order to construct large and complex software systems which provide the necessary infrastructure in a systematic manner, the focus in the development methodologies has switched in the last two decades from functional issues to structural issues. The encapsulation principle is essential to both the object-oriented and the more recent component based software engineering paradigms. Formal methods have been applied successfully to the verification of medium sized programs in protocol and hardware design. In this paper a brief review about software systems and essential survey of formal methods has been presented.

Index Terms— Design Patterns, Formal Specification, Formalisation, Formalism, Patterns Languages, Software Architecture, Software Process.

1 Introduction

As software systems become more complex, the overall system structure—or software architecture—becomes a central design problem. A system's architecture provides a model of the system that suppresses implementation detail, allowing the architect to concentrate on the analyses and decisions that are most crucial to structuring the system to satisfy its requirements.

One of the most challenging tasks in software development is to assure reliability of systems being designed and constructed. This becomes even more important as the use of software increases dramatically in embedded systems within life-critical environments such as medicine, air traffic control and other transportation systems, spacecraft control, and national defense weapons deployment and activation.

Recent research is demonstrating the clear advantages of a more formal and mathematical approach to software requirements capture and design. Methods used in such an approach are collectively called formal methods for software specification, and these methods have been shown to provide added reliability by modeling requirements in a way that they can then be reasoned about in a rigorous and repeatable manner [8],[9]. In general, the term formal methods refer to the use of techniques employing formal logic and discrete mathematics in the specification, design, and implementation of software (and hardware) systems.

The formal world of software engineering is closely connected to mathematics, in particular to mathematical logic and algebra. It tries to build up a mathematical theory and a calculus to deal with programs and requirements specifications in the style of a mathematical derivation. In the formal world, any document has to obey a precisely defined syntax, and also the semantics of documents is defined with mathematical precision. This is possible if the syntax has semantics in terms of another mathematical formalism, or if a calculus of deduction rules has been defined for the language under consideration, or both. Programming languages already provide formal specification for the logic described to solve a problem. [3]

Formal methods involve a high degree of mathematical formalism, and hence require a corresponding degree of commitment on the part of the learner to achieve a level of comfort approaching that most software developers have with traditional requirements analysis methods, with their dependence on English-like specifications.

Unfortunately, current representations of software architecture are informal and ad hoc. While architectural concepts are often embodied in infrastructure to support specific architectural styles and in the initial conceptualization of a system configuration, the lack of an explicit, independently-characterized architecture or architectural style significantly limits the benefits of software architectural design in current practice.

2 Patterns

2.1 The Fundamental Role of Patterns

Patterns are an important part of today's software engineering practice. They are a proven way of capturing working solutions to recurring problems, including their applicability, trade-offs and consequences. So how do patterns factor into the approach described above?

Architecture Patterns and Pattern Languages describe blueprints for architectures that have been used successfully. They can serve as an inspiration for building you own system's architecture. Once you have decided on using a pattern (and have adapted it to your specific context) you can make concepts defined in the pattern first class citizens of your DSL. In other words, patterns influence the architecture, and hence the grammar of the DSL.

Design Patterns, as their name implies, are more concrete, more implementation-specific than architectural patterns. It is unlikely that they will end up being central concepts in your architecture DSL. However, when generating code from the models, your code generator will typically generate code that resembles the solution structure of a number of patterns. Note,

- *A.V.Sriharsha, is currently pursuing PhD in Computer Science and Engineering, SV U College of Engineering (Autonomous), S V University, Tirupati, India, E-mail: avsriharsha@yahoo.com.*
- *Dr. A. Rama Mohan Reddy, is Professor in Department of CSE, SV U College of Engineering (Autonomous), S V University, Tirupati, India. E-mail: ramamohansvu@yahoo.com*

however, that the generator cannot decide on whether a pattern should be used: this is a tradeoff the (generator) developer has to make manually.

2.2 Pattern Languages

Christopher Alexander's work is based on the premise that the quality without a name is an objective characteristic of things and places. As an architect, Alexander wants to know where this quality comes from and, more important, how to create it, how to generate it. In the previous essay, "The Quality Without a Name," we learned of the divorce centuries ago of beauty from reality. That science could survive the divorce is understandable because science seeks to describe reality. Science can live and succeed a long time before it needs to concern itself with describing what makes something beautiful— when something is contingent, as beauty seems to be in modern science, there is little need to describe it. Art, on the other hand, cannot ignore beauty or the quality without a name because artists create things—paintings, sculptures, buildings—that are beautiful, that have the quality without a name. There are few fields that blend art and science: Architecture is one, and computer science is another. Architects must design buildings that can be built and architects have a "theory" about what they do—at least architects like Alexander do. In computer science we can describe theories of software, and we create software.

2.3 Architecture Complexity

Architecture is typically either a very non-tangible, conceptual aspect of a software system that can primarily be found in Word documents, or it is entirely driven by technology. An important problem facing software developers is the increasing size and complexity of software systems. As the expectations of users of software increase, software developers are expected to produce software to handle more difficult problems on a larger scale. As the complexity of software systems increases, the overall system structure—or software architecture—becomes a central design problem. Software architecture provides a model of the large scale structural properties of systems. These properties include the decomposition and interaction among parts as well as global system issues such as coordination, synchronization, and performance.

The software architecture of a system often appears in system descriptions as a "boxes and lines" diagram. This diagram structures the system in terms of particular kinds of computations and their composition. For example, the architecture of a payroll system might decompose it into three parts: a database, a report generator, and a data entry front end. These parts appear as boxes in an architectural diagram. Lines connecting them indicate the use of queries and updates supported by the database.

Software architecture raises the level of abstraction at which developers can reason about their systems. A system's architecture provides a model of the system that suppresses implementation detail and increases the independence of system components, permitting many issues to be localized. By suppressing these details at the architectural level, the architect can concentrate on the analyses and decisions that are most crucial to the system structure.

A critical issue in software architecture is composition. Once a system has been decomposed into components, they must be re-composed to define the structure of the system as a whole. An important class of composition in software architecture is active interaction between components based on discrete actions. Components each carry out some part of the total computation and interact to combine their behaviors, resulting in a behavior for the system as a whole. Interactions can be quite simple, such as in a batch model where each component acts separately, one executing to completion, its output providing the input to another component, which executes in a separate phase. Interactions can also be quite complex, such as network protocols of distributed systems, where each component can initiate communication, generate messages, and respond to other components' messages, where buffering, reliability, and authentication of information passed between components must be taken into account.

2.4 Architecture Style

Another important aspect of software architecture is the extension of design to exploit commonalities across families of systems. When developing a particular system, designers tend not to explore all possible alternatives for its architecture. Instead, they use specific patterns and idioms that are effective for the domain in which they are working. These patterns and idioms constrain the design space, permitting developers to ignore complications and alternatives that are not relevant to the system that they are developing. This exposes the issues that are most important and thus helps the developer make effective choices and locate the best solution more easily. We term such a collection of patterns and idioms an architectural style. Using a style has many benefits. A style focuses the design problem on techniques that are effective for a specific class of systems. By recognizing that, for example, real-time considerations are not of interest to a payroll database, developers can instead concentrate on developing a flexible and general set of queries for the database. A collection of components and connectors that work within a style enhances flexibility and reuse. The use of particular models supports higher-level design abstractions. If a style guarantees that a set of properties hold, it can lead to more powerful analyses than a general architecture permits.

2.5 Problems with Existing Architectures

Unfortunately, with few exceptions current exploitation of software architecture and architectural style is informal and ad hoc. While architectural concepts are exploited in infrastructure to support architectural styles and in the initial conceptualization of a system configuration, the lack of an explicit, independently characterized architecture or architectural style significantly limits the extent to which software architecture can be exploited using current practices. Currently, architectural configurations are typically described using informal box and line diagrams in design documentation, providing little information about the actual computations represented by boxes, their interfaces, or the nature of the interactions between them.

2.6 The Need for a Theory of Architectural Connection

Large software systems require decompositional mechanisms in order to make them tractable. By breaking a system into pieces it becomes possible to reason about overall properties by understanding the properties of each of the parts. Traditionally, Module Interconnection Languages (MILs) and Interface Definition Languages (IDLs) have played this role by providing notations for describing (a) computational units with well-defined interfaces, and (b) compositional mechanisms for gluing the pieces together. A key issue in design of a MIL/IDL is the nature of that glue. Currently the predominant form of composition is based on definition/use bindings.

In this model each module defines or provides a set of facilities that are available to other modules and uses or requires facilities provided by other modules. The purpose of the glue is to resolve the definition/use relationships by indicating for each use of a facility where its corresponding definition is provided. This scheme has many benefits. It maps well to current programming languages, since the kinds of facilities that are used or defined can be chosen to be precisely those of an underlying programming language. (Typically these facilities support procedure call and data sharing.) It is good for the compiler, since name resolution is an integral part of producing an executable system. It supports both automated checks (e.g., type checking) and formal reasoning (e.g., in terms of pre- and post-conditions). And, it is in widespread use.

However, the problem with this traditional approach is that, while it is good for describing implementation relationships between parts of a system, it not well-suited to describing the interaction relationships that occur in architectural abstractions.

The distinction between a description of a system based on "implements" relationships and one based on "interacts" relationships is important for three reasons. First, the two kinds of relationship have different requirements for abstraction. In the case of implementation relationships it is usually sufficient to adopt the primitives of an underlying programming language – e.g., procedure call and data sharing.

In contrast, as noted earlier, interaction relationships at an architectural level of design often involve abstractions not directly provided by programming languages: pipes, event broadcast, client-server protocols, etc. Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system. Hence, the abstractions associated with interactions reflect diverse and potentially complex patterns of communication. Second, they involve different ways of reasoning about the system. In the case of implementation relationships, reasoning typically proceeds hierarchically: the correctness of one module depends on the correctness of the modules that it uses. In the case of interaction relationships, the components (or modules) are logically independent of each other: the correctness of each module is independent of the correctness of other modules with which it interacts. Of course, the aggregate system behavior depends on the behavior of its constituent modules and the way that they interact. Third, they involve different requirements for compatibility checking. In the case of implementation relationships, type checking is used to determine if a use of a facility matches its definition. In the case of interaction relationships, we are more interested in whether protocols of communication are respected. For example, does the reader of a pipe try to read beyond the end-of-input marker; or is the server initialized before a client makes a request of it.

3 FORMALISMS

3.1 What is Formalism?

Formalism is mathematics. Mathematicising a system with deterministic set of input and objectives. Formalism of a system is a progressive predictive model. Transcripting a system into complex syntactic notations is formalism. Formula, Formal Specification, Formal Model are phase wise development of transcripted system. A formal language can be used to design a formal model of a system. Formula describes a mathematical transcription of an operation. Formal specification describes a homogenous and related set of operations which can be later described as a module of a system. A formal model is a module conglomerate, which describes the entire system.

3.2 Importance of Formal Representation

Accurate and complete requirements specifications are crucial for the design and implementation of high-quality software. Unfortunately, the articulation and verification of software system requirements remains one of the most difficult and error-prone tasks in the software development lifecycle. The use of formal methods, based on mathematical logic and discrete mathematics, holds promise for improving the reliability of requirements articulation and modeling. However, formal modeling and reasoning about requirements has not typically been a part of the software analyst's education and training, and because the learning curve for the use of these methods is nontrivial, adoption of formal methods has proceeded slowly.

3.2 Formal Specifications

Formal specifications use mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved. They describe what the system must do without saying how it is to be done. This abstraction makes formal specifications useful in the process of developing a computer system, because they allow questions about what the system does to be answered confidently, without the need to disentangle the information from a mass of detailed program code, or to speculate about the meaning of phrases in an imprecisely-worded prose description.

4 SURVEY

4.1 Conceptual Limits

The survey is just not limited to the goals and objectives of this paper. Right from the evolutionary deduction of patterns from the nature to the deterministic idealogical composition of patterns fitting to a rationale, the survey should be carried out for all sorts of pattern and pattern families.

The *raison d'être* of formal methods is to support rea-

soning: reasoning about hardware and software, reasoning about their properties and about their construction [4]. Of particular importance is the ability to reason, mathematically, about properties that are required for all values from unmanageably large data sets.

4.2 Formal Specifications

The concept of Formalization in Software has put forth practically by Charles Rich and Richard C. Waters, in their work "Formalizing Reusable Software Components", in the Artificial Intelligence Laboratory of Massachusetts Institute of Technology, July 1983. The intention of formalizing software begins with denoting the reusable libraries. Libraries are built in a platform for developing software, unlike if the need of functionally specified modules increase and their existence in the standard libraries diminish the collection of such modules starts and where are these preserved is the question of the time. Collecting and preserving such components for reuse.

In the context of formal methods, there are two important and yet memorable contributions that has eliminated myths on formal methods, viz., Seven Myths of Formal Methods [5] and Seven More Myths of Formal Methods [6]. Many industrial and research myths about formal have been dispelled by the authors that prevail in modeling and design based on their observations. The first seven myths of [5] challenge on the critical software system design and some traditional myths regarding the cost of development. But when these are merged with the problems of optimization the software analysis, design and development takes more swift steps, yet unacceptable present software engineers. The later seven myths stand iconoclast challenging the indispensability of formal methods in software development.

Thus formal methods contribute the basic platform or language for representing the software when it is not yet produced. In many applications, analysis and design lack transparency, which can be overcome with formal representation, with an outfit of excellent reliability analysis. Basically, a software designer believes strong in the non-functional specifications of the project, when formal methods are in light, it would be very easy to assess the characteristics of the software in the pre-development stage mathematically.

As quoted in [7],[8],[9] there are methods available for modeling web navigations, knowledge based analysis, feature models.

4.3 Tool Support

Software Tools are indigenous efforts for analysis and design. Rational RoseTM, is provides a wide elliptical palette of options and operations that enable a learner designer to design even a large scale software solution. With its ancillary applications, has obtained a wide publicity of using it right from education, development and research. The unified process of software development gives an in the tool that can mathematically convince the end user about his problem. Other tools exists for specific support of formal method specifications in the software analysis and design, such as `D-Finder 2`, Infer, `OpenJML`, `opal`, provide a varied applications for incremental design, testing and model checking [10],[11],[12].

5 Conclusion

In this paper we have found that formal mechanism is a very essential and vital process of understanding the quintessence of the software product development problem. The software architecture as seen in the previous decade is not at the apprehensible levels of a software team, it has so many desiderate and conglomerate patterns that should be digested during analysis and design of software product. The formal methods not only provide the mathematical path for analysis design but also for modeling, verification and checking, testing.

References

[1] Robert Allen, David Garlan, "Formal Connectors", March, 1994, CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[2] J. M. Spivey, "The Z Notation: A Reference Manual", Second Edition, Programming Research Group, University of Oxford. ©1992.

[3] Heinrich Hußmann, "Formal Foundations for Software Engineering Methods", ISBN 3-540-63613-7 © Springer-Verlag Heidelberg, 1997.

[4] John Cooke, "Constructing Correct Software", ISBN 1-85233-820-2, © Springer-Verlag London Limited 2005.

[5] Anthony Hall, Praxis Systems, "Seven Myths of Formal Methods", IEEE Computer Society, © 1990.

[6] Jonathen P. Bowen and Michael J. Hinchey, "Seven More Myths of Formal Methods", IEEE Computer Society © 1994.

[7] Jessica Chen, Xiaoshan Zhao, "Formal Models for Web Navigations with Session Control and Browser Cache", In Proceedings 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004.

[8] Kai Baukus, Ron van der Meyden, "A Knowledge Based Analysis of Cache Coherence", In Proceedings 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004.

[9] Wei Zhang, Haiyan Zhao, Hong Mei, "A Propositional Logic-Based Method for Verification of Feature Models", In Proceedings 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004.

[10] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, Rongjie Yan, "D-Finder 2: Towards Efficient Correctness of Incremental Design", In the Proceedings NASA Formal Methods, Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011.

[11] Cristiano Calcagno, Dino Distefano, "Infer: An Automatic Program Verifier for Memory Safety of C Programs", In the Proceedings NASA Formal Methods, Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011.

[12] Andreas Engelbredt Dalsgaard, Ren´e Rydhof Hansen, Kenneth Yrke Jørgensen, Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, Ji˘ri´ Srba, "opaal: A Lattice Model Checker", In the Proceedings NASA Formal Methods, Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011.

[13] Richard P. Gabriel, "Patterns of Software: Tales from the Software Community", OXFORD UNIVERSITY PRESS, (C) 1996, ISBN 0-19-5100269-X..