Design for Verification: Using Design Patterns to Build Reliable Systems[1]

Peter C. Mehlitz
CSC, NASA Ames Research Center
pcmehlitz@email.arc.nasa.gov

John Penix
NASA Ames Research Center
John.J.Penix@nasa.gov

Abstract

Components so far have been mainly used in commercial software development to reduce time to market. While some effort has been spent on formal aspects of components, most of this was done in the context of programming language or operating system framework integration. As a consequence, increased reliability of composed systems is mainly regarded as a side effect of a more rigid testing of pre-fabricated components.

In contrast to this, Design for Verification (D4V) puts the focus on component specific property guarantees, which are used to design systems with high reliability requirements. D4V components are domain specific design pattern instances with well-defined property guarantees and usage rules, which are suitable for automatic verification. The guaranteed properties are explicitly used to select components according to key system requirements.

The D4V hypothesis is that the same general architecture and design principles leading to good modularity, extensibility and complexity/functionality ratio can be adapted to overcome some of the limitations of conventional reliability assurance measures, such as too large a state space or too many execution paths.

1. Introduction

High dependability systems can be characterized by the need to satisfy a set of key properties at all times. This includes standard properties like absence of deadlocks or constant space execution, and application specific properties such as guaranteed responses or "correct" results. General approaches to demonstrating compliance with these properties are testing and formal verification.

Testing has inherent limitations with respect to non-reproducible execution environment behavior (like thread scheduling), which can be regarded as non-determinisms that are not fully controllable in the test environment. Thus testing covers only a small fraction of the potential state space of concurrent applications.

If higher confidence is needed, formal verification methods like model checking [1] or static analysis can be used. However, due to the inherent state space explosion, these methods tend to not scale well, which usually results in having to manually create models of the system to check. This process introduces potential fidelity problems. Moreover, it can also be so expensive that verification becomes a one time effort, which is inconsistent with the evolutionary nature of large systems development. The difficulty in verifying formal properties in turn often leads to a lack of properties in the system specification, creating additional fidelity problems by having to guess the verification goals.

No matter if testing or verification is used, both require a co-operative program design. For testing, design choices mainly determine the achievable test granularity (unit tests). Verification depends on a suitable program design for applicability of its modeling techniques (e.g. for abstraction). This leads to the implication of explicitly using appropriate design measures, instead of compensating their lack by means of tools and modeling techniques.

A general approach to the verification of large systems therefor is to use composition to build a system from separately verifiable parts,. This is the approach followed by Design for Verification (D4V), based on the assumption that the same design principles can be used not only to increase verifiability, but also to help testing, and esp. to improve understanding and extensibility of the target system.

[1] The research described in this report was performed at NASA Ames Research Center's Automated Software Engineering group and is funded by NASA's Engineering for Complex Systems program.

2. The Design for Verifiability Approach

The D4V components are not classical modules. Object oriented designs typically use a mix of inheritance (for static variation) and delegation (for runtime variation). The application mainly provides parts that are hooked into a usually much bigger framework library. The most abstract model for this is not the (language specific) class model, but sets of collaborating types with dedicated roles. This is essentially what came to be known as Design Patterns [2].

So far, Design Patterns have mainly be used as "mental building blocks". They come with various degrees of collaboration details, ranging from high level architectural patterns (not explicitly naming interfaces or aggregates) down to language specific idioms (coding patterns at expression level). Since a primary quality of a design pattern is its genericity, i.e., how readily it can be applied to a range of similar concrete problems, patterns often come with a deliberate lack of formalism, to leave enough freedom for problem-specific implementations. This otherwise helpful simplicity can make it difficult to use automated checks for correct pattern implementation and usage, which is on the other hand required to deduce properties for a target system composed of certain patterns. Bridging this gap between human-oriented fuzziness and tool oriented formalism is the major challenge for the D4V approach.

Ultimately, D4V strives to support the design process at two different levels:

- domain specific pattern systems
- aspect oriented implementation

The first level provides the building blocks from which to compose systems, the second level gives guidelines for how to implement these components.

Our initial target domain are state machine based, observable robotics applications with asynchronous transition triggers (events).

2.1 Domain Specific Pattern Systems

The D4V pattern systems consist of application domain specific libraries with static pattern components, plus a lookup schema to identify suitable patterns.

Each pattern instance comes with a set of guaranteed properties and a set of formal rules how to use the pattern so that the guarantees will hold.

The guarantees form the premier selection criteria for pattern lookup, which constitutes the main principle of D4V – to choose components based on verifiable properties derived from key requirements. We do not design a system and later on try to find out what properties we can check by means of existing verification tools, but rather design the system based on what we want to verify.

Examples for such properties could be a asynchronous event multiplexer (EventQueue) component which guarantees non-blocking, constant time multiplexing and prioritized event retrieval.

The usage rules mainly refer to implementation constraints of application provided types used in a pattern instance. The nature of these rules has to be formal enough to enable automated checks. It is essential to note that usage rule checks can be applied in the same fashion like regression testing. Since the property guarantees of a given pattern are invariant, verification can be turned into a automated development co-process.

A typical example is a *Observer* pattern variant which guarantees that all observers of a *Subject* will be synchronously notified whenever the *Subject* changes state. A usage rule would be that *Observers* do not perform any potentially unbound blocking operation from inside of their notification action, since this would prevent all subsequent *Observers* from getting notified. The check itself could be performed with static analysis.

Beyond this focus on safe implementation of "essential system complexity", there is also a important side effect of reducing harmful "accidental complexity" [3], which is a typical outcome of adding features to systems which were not designed for extensibility. To quantify this aspect, we have taken a small. moderately object-oriented. autonomous robot application and re-designed it using standard design patterns.

	Original version	D4V version
classes	82	37
interfaces	1	10
NCLOC	5926	1745
max WMC	397	56
sum WMC	1426	389
threads	6	2

Both systems were written in Java. WMC stands for "Weighted Methods per Class", the sum of the cyclomatic complexities of its methods.

The pattern oriented re-design not only resulted in the anticipated extensibility and test-suitability, especially for unit tests, but also showed a significant reduction in overall size, and an elimination of the complexity "hot spots" (max WMC). Just the decrease in threads alone makes the system more understandable, less error-prone (deadlocks), and more verifiable (state space).

2.2 Aspect oriented Implementation

D4V focuses on three essential aspects [4] of its pattern implementations, each one being represented by explicitly marked and annotated code sections:

- consistent program states (CheckPoints)
- conceptual branches (BranchPoints)
- potential extensions (ExtensionPoints)

(a) *CheckPoints* are locations where required-to-be consistent states which are relevant to component property guarantees have to be checked. This includes freely placeable assertions as well as pre-, post-conditions and invariants. The checks themselves can refer to explicit program state (variable values) and implicit execution environment state (number of instructions, relative time etc.). Checkpoints are linked to their corresponding property guarantees. The concept is closely related to "programming by contract", and basically defines the underlying correctness model of a component implementation. While evaluation of check points is straightforward (provided the programming environment has a assertion mechanism, reachability analysis and side-effect detection of check points is again subject to tool support.

A typical example is a check for memory leaks after a certain operation has been completed, to verify constant-space execution properties.

(b) *BranchPoints* denote locations that are relevant for both testing and model checking. Only conceptual BranchPoints are identified, not every branch in the control flow. We include only operations that are non-deterministic from the applications perspective, esp. potential context switching operations in multi-threaded programs (e.g. Thread starts, locking attempts, blocking I/O). Every BranchPoint has a description of it's possible choices and their corresponding conditions. For testing, the BranchPoints describe the required coverage, and hence form the basis for (automated) test case generation. For model checking, they can be considered as the "built-in model" providing potential backtracking targets and atomic sections.

We are investigating program designs that use explicit ChoiceGenerator objects in BranchPoints, with the goal to enable "in-situ" model checking by means of effectively turning threads into co-routines (i.e. making programming environment specific non-deterministic actions reproducible). A welcome side effect of this approach would be that systems can be tested and verified in their real target environment, instead of the environment which is required to run complex tools.

Such a BranchPoint implementation scheme also reflects the observation that (a) concurrent systems should be designed around their synchronization/communication points, and (b) these operations are usually encapsulated into APIs or specific language constructs anyway (i.e. can be easily intercepted).

(c) *ExtensionPoints* identify the locations that are relevant for extending the applications functionality without breaking its design. They include potential base classes and interfaces with their corresponding variations (e.g. overridable methods), specifying associated implementation constraints.

The reason why we focus on this aspect in the D4V context is the fact that the development of a complex system is hardly ever completed [5]. The typical case is a evolutionary extension of functionality, which can easily lead to accidental complexity and feature bloat, violating properties which did hold in the original design. The goal is to enable effective assessment of the effects potential feature extensions might have on reliability requirements.

3. Project Status and Outlook

The D4V project is still in its early stages. The current focus is on the development of a suitable design pattern system based on our motivating example, a event driven, observable, state-model based control system for autonomous robots. We plan to eventually have three different versions of the system as a basis for metrics comparison

- the original version exposing typical effects of accidental complexity
- the standard design-pattern implementation to show the reduction of complexity and increase of extensibility
- a version that uses D4V specific patterns to show the property-guarantee driven design process

This approach reflects our view that D4V is not a radically new design methodology, but rather extends and combines already accepted "best design practices" in order to overcome the traditional gap between design/development and testing/verification, which causes not only problems for finding defects, but also for subsequently fixing them.

References

[1] W. Visser, K. Havelund, G. Brat, S. Park. "Model Checking Programs", Proceedings of the 15th International Conference on Automated Software Engineering (ASE), Grenoble, France, September 2000.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - “Design Patterns Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995

[3] Frederick P. Brooks - “No Silver Bullet: Essence and Accidents of Software Engineering”, Proceedings of the IFIP '86 conference

[4] Tzilla Elrad, Robert Filman, Atef Bader - “ Aspect Oriented Programming”, CACM Vol 44 No. 10, October 2001

[5] David Parnas - “Designing Software for Ease of Extension and Contraction”, IEEE Transactions on Software Engineering, SE-5(2):128--38, Mar. 1979.