Design for Verification: Enabling Verification of High Dependability Software-Intensive Systems[1]

Peter C. Mehlitz
Computer Sciences Corporation
pcmehlitz@email.arc.nasa.gov

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

Lawrence Z. Markosian
QSS Group, Inc.
lzmarkosian@email.arc.nasa.gov

Abstract

Strategies to achieve confidence that high-dependability applications are correctly implemented include testing and automated verification. Testing deals mainly with a limited number of expected execution paths. Verification usually attempts to deal with a larger number of possible execution paths. While the impact of architecture design on testing is well known, its impact on most verification methods is not as well understood. The Design for Verification approach considers verification from the application development perspective, in which system architecture is designed explicitly according to the application's key properties.

The D4V hypothesis is that the same general architecture and design principles that lead to good modularity, extensibility and complexity/functionality ratio can be adapted to overcome some of the constraints on verification tools, such as the production of hand-crafted models and the limits on dynamic and static analysis caused by state space explosion.

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, and application specific properties such as guaranteed responses or "correct" results.

Testing at various "scope levels" is usually the preferred way to check deterministic computation results, but this approach is of limited value for checking properties of concurrent programs. Since the scheduling behavior typically cannot be controlled from the testing environment, standard defects like race conditions and deadlocks can easily be missed by testing. This is an important case in which automated system verification comes into play.

If verification is left until system integration, the target system often is already too big and complex for verification tools to handle it directly, which is especially true for static analysis and model checking of concurrent programs. As a result, target systems need to be modeled in order to apply the tools, an expensive process that also has the potential for introducing fidelity problems. Because of the associated costs, model-assisted verification can easily degenerate into a one-time-effort, which simply does not match the evolutionary life cycle of large systems. Lack of support for efficiently checking formal properties in turn leads to minimal inclusion of such properties in the specification and design phases, which further decreases the effectiveness and value of automated verification.

The *Design for Verification* (D4V) hypothesis is that the same general architecture and design principles that lead to good modularity, extensibility and complexity/-

[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.

functionality ratio can be adapted to overcome some of the constraints on verification tools.

The context of our D4V work is the development of practical tools and methodologies based on source code model checking technologies[2] such as Java PathFinder[1], but the D4V concepts are intended to be applicable with a broad range of verification approaches.

2. Traditional Approaches

One verification approach is based on architecture design documentation (e.g. UML/OCL), with the intent of producing correct architectures, from which code is more likely to correctly implemented.

Another approach to overcoming the scalability problem for verification tools is to improve the tools. In the case of model checking, this involves techniques like abstraction, slicing and partial order reduction. These are necessary techniques for handling real applications.

Even with these sophisticated approaches, it still is too easy to design applications so that they cannot be applied. This is particularly due to the fact that in contemporary programming environments, an increasing amount of functionality is shifted from stand-alone applications into libraries and frameworks, which either exceed the size constraints of the verification tools, are or unavailable in a suitable format to apply these tools.

3. The Design for Verifiability Approach

Our approach complements traditional approaches in that we explicitly add verification- and testing-specific considerations to the architecture design phase. The general idea is to map key requirements of the specification directly to dedicated, mostly invariant design components, which can be verified separately. The goal is to turn system verification into a development co-process like regression testing.

We try to achieve this goal by using domain specific design pattern collections. Each pattern instance comes with a set of formal usage rules and guarantees. Usage rules are subject to automated checks, mostly using contracts (preconditions, post-conditions, and invariants) and static analysis. The pattern selection process itself is driven by evaluation of the guaranteed properties against the key specification requirements. While this does not ensure arbitrary, application-specific properties, it gives a much better understanding of the formal correctness model early in the development phase.

Since these key patterns constitute design elements that are mostly invariant during the implementation and evolution of the system, the verification results are not lost, and the tools can be re-applied at later stages of the system lifecycle without modeling efforts.

The program design is centered around three concepts: *extension points*, *conceptual branch points*, and *check points*.

Extension points identify the components that can be used to extend the functionality of the application without breaking its design or causing feature bloat. Extension points include potential base classes with their overridable methods, and major delegation objects with their associated interfaces, both with their corresponding implementation constraints. Extension points allow property verification during later stages of the lifecycle, when system functionality is often extended without having a suitable design infrastructure for these extensions.

Conceptual branch points are the locations that are relevant for both testing and model checking. This includes non-deterministic operations, in particular potentially blocking or context switching instructions in multi-threaded programs, which are preferred targets for backtracking. We are investigating program designs that turn these branch points into choice generator calls, enabling systematic testing and model checking in the real execution environment. This is achieved by turning implicit, execution environment specific behavior (like thread scheduling) into explicit delegation objects (the generators). To verify multi-threaded programs, this can be used to effectively turn threads into co-routines, which are systematically switched inside of the generator objects. This approach is based on the assumption 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).

Check points describe the application-specific correctness model, and map to freely-placeable assertions. They can be thought of as required-to-be consistent, usually global states, and should be mappable from/to the system specification. A typical example is a check for memory leaks after a certain operation has been completed, to verify constant-space execution properties. 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.

It is important to note that D4V does not attempt to introduce a radically new design approach, but instead

[2] *Source code model checkers* take the source code of an application (or some transformation of it) as the model. Examples include SPIN and SLAM for C, and Java PathFinder for Java

extends existing "best design practices" towards verifiability and testability. This comes with two intentional side effects.

First, deliberate use of design patterns tends to improve modularity and reduce "accidental complexity". This in general makes the system more understandable and unit-testable, and reduces the relevant state space for verification tools.

To quantify this aspect, we have taken a small. moderately object-oriented. autonomous robot application and re-designed it using design patterns.

	old version	***new 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" and represents the sum of the cyclomatic complexities of its methods.

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

Second, D4V attempts to overcome the traditional gap between design/development and testing/verification. Because designers gain more scalable tools and tests, they are encouraged to think more about application correctness.

4. Project status

The D4V project is in an early stage. The current focus is on the development of a suitable design pattern system. Our first target domain is event driven, observable, state-model based systems.

[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.