A Case for Test-Code Generation in Model-Driven Systems

Matthew J. Rutherford and Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, Colorado, 80309-430 USA
{rutherfo,alw}@cs.colorado.edu

Abstract. A primary goal of generative programming and model-driven development is to raise the level of abstraction at which designers and developers interact with the software systems they are building. During initial development, the benefits of abstraction are clear. However, during testing and maintenance, increased distance from the implementation can be a disadvantage. We view test cases and test harnesses as an essential bridge between the high-level specifications and the implementation. As such, the generation of test cases for fully generated components and test harnesses for partially generated components is of fundamental importance to model-driven systems. In this paper we present our experience with test-case and test-harness generation for a family of model-driven, component-based distributed systems. We describe our development tool, MODEST, and motivate our decision to invest the extra effort needed to generate test code. We present our approach to test-case and test-harness generation and describe the benefits to developers and maintainers of generated systems. Furthermore, we quantify the relative cost of generating test code versus application code and find that the artifact templates for producing test code are simpler than those used for application code. Given the described benefits to developers and maintainers and the relatively low cost of test-code development, we argue that test-code generation should be a fundamental feature of model-driven development efforts.

1 Introduction

A primary goal of generative programming and model-driven development is to raise the level of abstraction at which designers and developers interact with the software systems they are building. During initial development, automatic generation of software artifacts from high-level specifications offers many advantages to system developers, including increased productivity, enhanced source-code consistency, high-level reuse, and improved performance of the generated system [2]. However, increased distance from system implementation can be a disadvantage during testing and maintenance phases. Testing must be performed to certify initial systems and to help cope with future changes, both planned and
unexpected; this does not change because the bulk of a system is automatically generated.

With model-driven systems, one might assume that framework and generated software have been debugged and certified elsewhere, and that testing the instantiated code is redundant and wasteful. While it is most likely true that framework and generated software have been tested, the question remains: in what context? In practice, any previous testing can be seen as irrelevant; the only context that truly matters is that of the particular system being created.

Testing artifacts serve as an essential bridge between the high-level specifications and the specific instantiation of a model-driven system. In the event of an error or failure, their existence provides a road map through the potentially vast amount of unfamiliar, generated code. Testing artifacts are particularly important in data-driven distributed systems, where software is deployed in potentially heterogeneous environments with complicated interactions across multiple tiers. In these systems, there are many layers surrounding the generated software that can be independently altered to conflict with the set of assumptions that were made at generation time. Although assumptions may be listed in the documentation of the abstract models or generators, the test cases (provided they give good coverage) are an executable form of these assumptions.

With an appropriate level of detail in the interface specifications of domain-specific components, it is possible to automatically generate some or all of their test cases. In fact, test cases can be specified and generated in parallel to the specification and generation of components. Of course, many model-driven systems are not completely generated. Instead, the “cookie-cutter” code is generated, and some crucial domain-specific components are hand written. Thus, these domain-specific components can only be fully tested by hand-coded test cases. Nonetheless, generation technology still has a role to play: In the same way that domain-specific components are constrained by the generated code surrounding them, test cases for these components are also constrained by generated code. The scaffolding that is generated to surround the domain-specific test cases is the test harness. Test harnesses are intended to handle as much test setup and cleanup as possible, allowing the developer to concentrate on the logic to perform the actual tests.

In this paper we describe our experience with a generative approach to test case and test-harness development. Collectively, we refer to test cases and harnesses as test code, and show how we generate the test code in parallel with the system it is meant to test.

Our experience is based on the use of a model-driven generative programming tool called MODEST (Model-Driven Enterprise System Transformer), developed by the first author while at Chronos Software, Inc\(^1\) All systems generated by MODEST have the same basic architecture and design. The systems differ in their domain-specific data and logic, and some features can be enabled and disabled, leading to generated variations on a basic theme. MODEST does not implement OMG’s Model Driven Architecture (MDA) standard \(^1\). However,