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

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 377–396, 2003.
© Springer-Verlag Berlin Heidelberg 2003

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 [12]. However,

[1] http://www.chronosinc.com/

there are enough similarities that many of the lessons learned could be readily applied to MDA-compliant tools.

In describing MODEST, we carefully distinguish among three roles: the developer, the customer, and the (end) user. A *developer* uses MODEST in the creation of a system tailored to the needs of a *customer*; the outcome of this activity is the structured system specification that serves as one of the inputs to MODEST. The developer repeatedly adjusts the code templates that serve as the other inputs to MODEST. Lastly, the developer implements domain-specific operations and the test cases for them. Once development is complete, the developer delivers both the application code and the test code to the customer. During maintenance, the customer has the ability to execute test code and adjust application and test code as needed. However, it is important to note that the customer does not have access to the generative capabilities of MODEST. Finally, the customer makes the system available to a *user* who interacts with the system at run time to achieve some business purpose.

MODEST represents a structured description of the system to be generated as an XML document. This document captures the domain-specific data model, the interfaces of domain-specific logic components, dependencies on third-party libraries, and characteristics of the deployed system. Artifact generation in MODEST is accomplished by a series of XSL transforms. Initially, the system specification is used to generate a customized build script for the entire system. This build script includes targets to generate all of the other artifacts that comprise the generated system. MODEST generates Java source code, database management scripts, and Enterprise JavaBean (EJB) deployment descriptors.

Systems developed using MODEST are intended to be delivered to customers who need to maintain and extend their system without access to MODEST's generative capabilities. This requirement has several far-reaching consequences, one of which, in fact, is the need to provide test code. To make this practical, the decision was made to try to generate as much of the test code as possible.

While the initial decision to provide test code was not made for technical reasons, the presence of the test code turned out to greatly enhance the development process of the underlying framework and the generation templates. In particular, by requiring template developers to think also about test-code generation, they became more familiar with the code being generated. Furthermore, when underlying infrastructure software (e.g., the database, application server, and the like) was changed, the test code enabled developers to quickly certify existing systems. On the other hand, there is a cost associated with the generation of test code. In this paper we attempt to quantify the complexity of code-generation templates, and compare the complexity of templates for test-code generation with templates for application-code generation. Given the 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.

In the next section we describe the important aspects of MODEST and the salient characteristics of the family of systems it generates. Section 3 explains

the strategy used by MODEST to generate test code. Section 4 presents an evaluation of our experience with MODEST, and also compares the complexity of test-code templates with application-code templates. Section 5 outlines related research, and Section 6 provides some concluding remarks.

2 An Overview of MODEST

MODEST is a model-driven code generation tool developed to streamline the programming activities of a small software consulting company. By generating "cookie-cutter" code from a structured representation of a customer's requirements, MODEST allows developers to concentrate on understanding the customer's needs, developing domain-specific logic, and adding to the corporate knowledge base. At the end of a development cycle, the customer receives delivery of a complete, self-contained system that satisfies their initial requirements. Obviously, this is an idealized version of the process, but the notion that the end product is self contained, ready for use or extension by the customer, is key.

Figure 1 depicts the three major conceptual elements of the tool: the domain specification, artifact templates, and the generative engine. Of these, the domain specification is the only customer-driven entity. From an engineering perspective, all systems generated by MODEST have the same architecture and design, which are embodied in the artifact templates. The variability of each system comes entirely from the different domain models that can be represented by the domain specification.

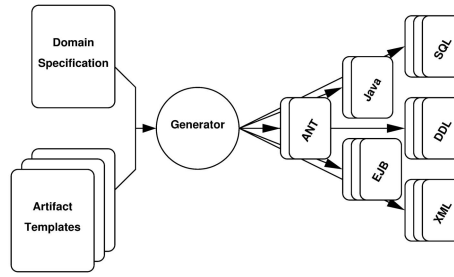


Fig. 1. Conceptual Elements of MODEST

The domain specification uses an XML document to represent a conceptual model of a customer's domain of interest, including the data model and the interfaces of any domain-specific business objects that are needed. The domain specification does not include engineering details, which are captured entirely by the artifact templates and the generator control script. The generator control script is itself created automatically.

The artifact templates are written in XSL, and are used by the generator to create different kinds of artifacts, including build scripts, Java source code,

Enterprise JavaBean deployment descriptors, and database creation and management scripts. During the generation of a given system, a particular template will be used separately for each instantiation of the artifact that it describes, as dictated by the system build script.

The generator is simply the Xalan XSLT engine wrapped inside an ANT task. It is controlled by ANT build scripts, the first of which contains the targets to generate the system-specific build script that, in turn, is used to control the generation of all other artifacts.

2.1 Domain Specification

Figure 2 contains a high-level view of the important sections of the MODEST domain specification. The figure is organized hierarchically to help convey the nesting of the XML elements. The domain specification is the only place within the MODEST environment that a customer's requirements are explicitly maintained.

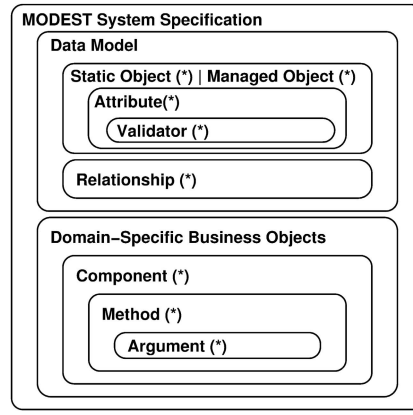


Fig. 2. Organization of the Domain Specification

The data model for the domain of interest is captured by specifying *static objects*, *managed objects*, their *attributes*, and the *relationships* among them. Static objects are those entities that are not intended to be changed by users of the system. Typically, static objects are used to model ancillary objects that exist to support the core domain entities. The attribute values for all instances of static objects must be detailed in the domain specification. The static object data are used to populate reference tables within the database, and flyweight pattern [5] classes are generated for use in the software.

Figure 3 shows a sample specification for the automobile domain. In this case, `Make` has been modeled as a static object with one attribute. Two instances of `Make` are available: `SUBARU` and `FORD`. The customer, after receiving the system generated from this domain specification, can add further instances.

```
<domain-specification>
  <static-object name="Make">
    <attribute name="name" type="String" unique="true" required="true">
      <validator type="string-length" min="1" max="64"/>
    </attribute>
    <instance name="SUBARU" key="10">
      <attribute-value name="name">Subaru</attribute-value>
    </instance>
    <instance name="FORD" key="10">
      <attribute-value name="name">Ford</attribute-value>
    </instance></static-object>
  <managed-object name="Car" type-key="20">
    <attribute type="String" name="id" unique="true" required="true">
      <validator type="string-length" min="1" max="128"/>
      <validator type="alphanumeric-string"/></attribute>
    <attribute name="make" static-object="Make" required="true"/>
  </managed-object>
  <managed-object name="Driver" type-key="30">
    <attribute type="String" name="name" required="true">
      <validator type="string-length" min="1" max="128"/>
      <validator type="alphabetic-string"/></attribute>
    <attribute type="Integer" name="age" required="true">
      <validator type="range" min="16" max="110"/>
    </attribute></managed-object>
  <relationship src="Car" dest="Driver" type="1-n"/>
  <business-object name="IdGenerator">
    <business-method name="nextId" return="String"/>
  </business-object></domain-specification>
```

Fig. 3. Sample Domain Specification

A *validator* in a domain specification is used to describe the range of values that are valid for a particular attribute. The validator details are used at run time to guard attribute values, and they are also used in the generation of data for unit and integration tests. In Figure 3, for example, a validator is given for the `name` attribute of the `Model` static object. For efficiency reasons, attribute values of static objects are not validated at run time, so development-time testing of instance attribute values is important. Furthermore, instances of static objects represent a software feature that has a high likelihood of being changed by customers after delivery, so the generated test cases are crucial to maintaining high-quality software.

Managed objects are those entities that can be created, updated, and deleted by the user during the operation of the system. Aside from the basic data types, managed objects can also have attributes of any declared static object types. Managed objects require more support code than static objects; since they represent persistent changeable data, they need to be stored and retrieved from a database, and they typically have relationships with other managed objects or static objects. Additionally, because attributes of a managed object are dynamic, the managed object must use validators to ensure that its internal state is always consistent.

The system modeled by the specification shown in Figure 3 contains the managed objects `Car` and `Driver`. For managed objects, the attribute validator properties are used to instantiate representative test data for the system. Values that would pass validation are used to test the functional operation of

the system, and invalid values are used to test error handling. Because managed objects represent the core persistent data processed by the system, test cases that exercise this aspect of the system are vitally important to the customer.

Relationship elements model a "has" relationship among managed objects. The three supported relationship cardinalities are one-to-one, one-to-many, and many-to-many. Figure 3 includes a one-to-many relationship between `Car` and `Driver`.

2.2 Generator

Code generation in MODEST is a straightforward instance of using XSL to transform XML documents into other XML documents and plain text files. The XML/XSL combination was used because the Chronos developers already had familiarity with the Xalan XSLT engine, and because decent tool support for creating XML documents already exists. Following the terminology of Czarnecki and Eisenecker [2], the MODEST artifact generator is a *transformational* generator that performs *oblique* transformations.

The structures that are used in the domain specification are significantly different from those that are embodied in the generated artifacts. In most cases the template that creates a particular artifact pulls data from many different parts of the specification. Figure 4 shows an augmented data-flow diagram for the MODEST code generator. The XSL transformer is controlled by two different build scripts, the *bootstrap control script* and the *generated control script*. The bootstrap script contains only the targets and dependencies needed to invoke the XSLT engine for the creation of the system build script. Once the system build script has been generated, it contains the dependencies and targets needed to generate all the other artifacts in the proper order.

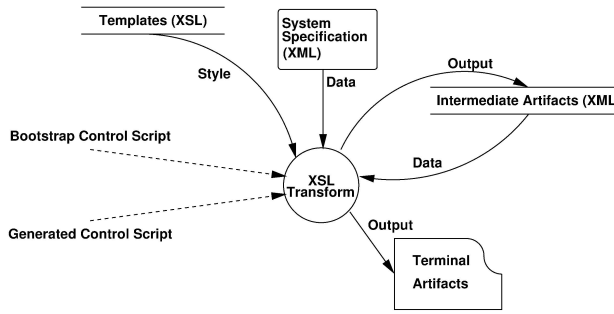


Fig. 4. Data Flow in MODEST

Several artifacts needed for the final system are generated using a multi-stage process. Figure 4 shows this as a distinction between *terminal artifacts* and *intermediate artifacts*.

2.3 Artifact Templates

All of the design and implementation decisions that go into the creation of the end product are codified in the artifact templates, which are XSL style sheets. There is a different artifact template for each type of artifact that is generated. For example, there is an artifact template that generates a Java class to encapsulate a managed object, and this template is parameterized by the intended name of the managed object. For a given system, a particular artifact template might be used multiple times, once for each instance of the artifact called for in the domain specification. The domain specification shown in Figure 3 contains two managed objects. Thus, the generated build script contains two transformations that use the managed-object template, one for `Car` and one for `Driver`.

2.4 System Family

From an engineering perspective, all systems generated by MODEST have the same design. Figure 5 provides a high-level view of the MODEST family of systems. Architecturally, the scope of MODEST is restricted to the application logic and data storage tiers of a standard three-tier architecture. The scope is further restricted in that MODEST systems are designed to operate within the Enterprise JavaBean (EJB) distributed object framework. This represents the target environment for code generation.

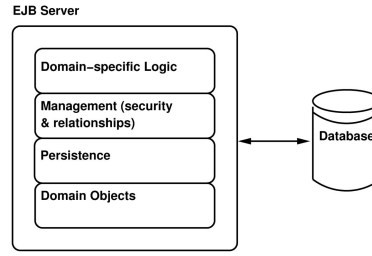


Fig. 5. MODEST System Family

The simplest classes generated by MODEST are those that encapsulate the attributes of managed and static objects. These simple classes are labeled "Domain Objects" in Figure 5.

The top three layers shown in Figure 5 all consist of EJB components. Both Java code and deployment descriptors are generated for these.

The components in the persistence layer are implemented as entity EJBs that contain the logic for storing and loading managed-object values to and from the database. There is a different entity EJB for each managed object in the domain specification. The persistence layer is not visible to external clients;

its functionality is only available to the management and domain-specific layers above it in the system architecture.

Next is the management layer whose objects are implemented as session EJBs. The objects perform relationship management, and authorization and authentication for clients. These components are used by client code to create, update, and delete managed objects as needed by the user. A separate EJB is generated for each managed object in the domain specification.

At the highest level in Figure 5 are the components that provide domain-specific logic. The interfaces for these components are described explicitly in the domain specification, and their implementation is performed manually by developers. However, deployment descriptors, remote interfaces, and framework code can all be generated directly from the domain specification. Framework code that allows these objects to interact properly with the security model is encapsulated in generated base classes through which all method calls are passed.

For example, consider the business object `IdGenerator` shown in Figure 3. This object has a single method with signature `String nextId()`. The remote interface for this object is as follows.

```
interface IdGenerator extends EJBObject
{
  String nextId( SecurityToken st ) throws AuthorizationException,
                                           ServerException,
                                           RemoteException;
}
```

The signature of the `nextId` method has been augmented with a standard parameter and standard exceptions. The generated base class contains the augmented method that is actually called by clients. This method wraps a call to the abstract method `nextIdImpl()` in code that checks the client's authorization and deals with logging and error handling in a standard way. The developer is responsible for providing a derived class that implements `nextIdImpl()` with the correct semantics.

By wrapping the methods of domain-specific logic objects in this way, MODEST ensures that security, logging, and error checking are handled in a consistent manner, allowing the developer to concentrate on the complexity of the business logic, not on the complexity of the MODEST framework. However, this scaffolding also makes it harder for the developer to test their hand-written logic directly, since they would have to understand quite a bit about the MODEST framework to be able to create an integration test case for it by hand. For this reason, it is critical that equivalent scaffolding be generated for the testing of the domain-specific logic.

Beneath all of the application logic sits the database. MODEST generates a schema creation script that can be used to create all of the tables and views needed to efficiently handle the data requirements of the managed and static objects that are outlined in the domain specification. Additionally, this creation script creates other database objects that are needed to enforce MODEST's security model and to validate data as much as possible.

3 Test-Code Generation

Test code is generated by the MODEST system in parallel with the main application code. Test code is generated for two major reasons.

1. To validate and certify development-time activities.

   Even given the generative capabilities of MODEST, development of distributed component-based systems is a complicated undertaking. Subtle, unexpected interactions among components, that are not present in simple domains, crop up in complicated domains. Additionally, implementation of a customer's domain logic is often complicated, and the existence of test harnesses and test cases reduces the chances that unexpected side effects will result from the hand-written portions of the system.

2. To provide a framework for supporting long-term maintenance activities.

   As described in Section 2, the systems generated by MODEST are delivered to customers under the assumption that maintenance and extension are going to be performed by customers without the benefit of MODEST's generative capabilities. The generation of test harnesses and test cases is an important piece of this business model, and it gives potential customers confidence that the software they receive is operational, and that it can be maintained and extended in a rational manner so that it remains operational.

In MODEST, test-code generation consists of four major aspects: generation of code to instantiate representative static and managed objects; generation of test cases for validating static-object implementations; generation of test cases for validating managed-object implementations; and generation of test harnesses for facilitating the testing of domain-specific logic.

3.1 Representative Object Instances

All test cases need access to representative data to exercise the functionality of the system. In MODEST, the lowest-level domain-specific classes are those that simply wrap the attributes of static objects and managed objects. In Section 2.4 these are referred to as "domain objects". Representative instances of these objects are needed throughout the generated test code for use as method parameters and the like. To accomplish this, a class that provides methods for instantiating representative domain objects is generated and used by other test cases.

For the domain specification shown in Figure 3, the base test-case class would have the following methods.

- `newMake()`: randomly chooses one of the `Make` instances.
- `newDriver()`: randomly populates attribute `name` based on its validators.
- `newCar()`: randomly populates attribute `id` using `newMake()` to pick a `Make`.

As mentioned above, the validators for managed objects are used as guidelines for choosing valid attribute values. The algorithms to select these values probabilistically generate null values, boundary condition values, and mid-range values to attempt to get adequate test coverage.

3.2 Test Cases for Static Objects

As mentioned in Section 2.1, static objects represent immutable data that play a supporting role to managed objects. They are implemented as reference tables in the database, and as flyweight pattern classes in software. Each static object type is supported by a factory that provides lookup methods and access to collections of flyweight classes. Both unit tests and integration tests are generated to validate their implementations. The static object `Make` contained in the domain specification shown in Figure 3 and its supporting factory, `MakeFactory`, is used as an example below.

Unit Tests. The implementations of static objects are validated with unit tests. All valid instances of static objects are enumerated in the domain specification, and these data are used to exhaustively test the software. Unit tests ensure the following properties.

1. All instances are present. This means that `Make.SUBARU` and `Make.FORD` are available and found in the `MakeFactory.all()` collection.
2. `equals()` and `compareTo()` work properly for each instance. The test cases would compare `Make.SUBARU` and `Make.FORD` to themselves and then to each other, and verify that the results were correct based on the data in the specification.
3. Attribute values match what is listed in the domain specification. The test cases would verify that `Make.SUBARU.getName() == "Subaru"`.
4. Lookup methods on factory classes work properly for each attribute of each instance. The test cases would verify that `MakeFactory.findByName( "Subaru" ) == Make.SUBARU`.

Integration Tests. Integration tests are needed to ensure that the generated implementation matches the data that are contained in the reference tables in the database. For efficiency purposes, there are no run-time checks of data consistency, so development-time checking is especially important. This is accomplished by selecting all the data from the database reference table, and ensuring that its contents match that of the `MakeFactory.all()` collection.

3.3 Test Cases for Managed Objects

Managed objects represent the core domain entities that comprise a particular domain. The managed-object instances are stored in the database, and their values can be changed, and instances can be deleted. Because they are the central entities in the system, and because they can be modified and deleted, they are guarded by a security model ensuring that a user is properly authenticated and authorized before a particular action is performed. There is code to deal with managed objects in three of the layers shown in Figure 5: domain, persistence, and management. Both unit tests and integration tests are needed to test the implementation effectively.

Unit Tests. Unit tests are generated to validate the domain-object implementation. These tests ensure that the validation code is working properly for each class by verifying that invalid attribute values cannot be used in constructors or mutator methods, and that randomly generated valid data can be used. There are also tests to ensure that standard methods such as `equals()` and `compareTo()` are implemented properly.

For the `Car` managed object shown in Figure 3, these tests would ensure that `Car` could not be instantiated without a valid value for the `id` attribute, and that it could not subsequently be changed to something invalid. The unit tests would also ensure that properly instantiated `Car` objects could be compared properly.

Integration Tests. Integration tests are generated to test the implementation of the persistence and management layers. The persistence layer implementation for each managed object is tested to ensure the following.

1. Managed-object data are replicated perfectly in the database after a store operation.
2. Managed-object data are replicated perfectly in the software after a load operation.
3. Two separately loaded copies of the same data compare properly.
4. Data are replicated perfectly in the database after an update operation.
5. Data are removed from the database after a remove operation.

Because of the design decision that managed objects cannot have invalid internal state, it is not necessary to test invalid data values at this level.

Integration tests are generated at the management level to ensure that relationships among managed objects are properly maintained. This involves ensuring that any required relationships are satisfied in the proper order and that invalid relationships are not permitted. For the `Car` object from Figure 3, this would mean ensuring that a valid `Driver` object existed before a `Car` was created, and that `Driver` objects could not be deleted if that would result in an unsatisfied `Car` relationship.

3.4 Domain Logic Test Harnesses

As described in Section 2, domain-specific logic is captured in the specification through business objects. The interfaces to these business objects are captured in the specification, but the semantics are not. Although the implementations of the actual business methods cannot be automatically generated, much of the scaffolding and supporting code can. For application logic, this means that the developer only has to concentrate on the complexity of the business rules and not the complexity of the MODEST framework. A similar approach is taken for the generation of test harnesses.

The goal of the generated test harness is to allow the developer to operate at the same level of detail at which the business method is implemented. This implies that the scaffolding code should handle any setup that is needed to interact

with the business method, and wrap the domain-specific testing logic properly to handle exceptions that are generated by the framework. For the `IdGenerator` business object shown in Figure 3, an abstract test case, `IdGeneratorTestCase`, would be generated that had the following methods.

- `setUp()`: performs authentication and authorization setup that is needed to have permission to call the business method.
- `nextId()`: proxy method used by the hand-written test code. Within this method, the actual call to the business method is performed with the appropriate security ticket and error handling.
- `tearDown()`: performs any clean up related to `setUp()`.

Developers are then responsible for extending this class and implementing the actual tests, using `nextId()` as a proxy for the remote EJB object.

4 Discussion and Evaluation

This section describes the benefits of test-code generation and quantifies its costs. Initially, we summarize some observations made during the development of the first few prototype systems. Next, we present an example in which generated test code is used to identify and debug a subtle integration problem. Finally, we analyze the relative complexity of creating test-code templates versus application-code templates.

4.1 Utility of Generated Test Code

This section contains some observations and analysis of the utility of test-code generation during development and maintenance activities.

Test Cases for Static Objects. The code generated to handle static objects is relatively simple. Since its initial development, few bugs have been found in the implementation. This is due partly to the simplicity of the code, and partly to the exhaustive nature of the generated test cases. For these reasons, static object test cases do not add a lot of value at development time.

Conversely, the real utility of the static-object test cases is to help ensure consistency during system maintenance. This is due to two factors: (1) a common way for customers to extend their system is to manually add new static-object instances for items that were overlooked initially and (2) proper implementation of the static objects requires that changes are made to two disconnected locations, the software classes and the database. The existence of exhaustive integration tests ensures that the two implementations are always consistent.

Test Cases for Managed Objects. The generated implementations of managed objects are fairly complicated and must be consistent across three different layers of the resulting system. These factors ensure that the generated test cases

are used often during system development to track down subtle bugs and inconsistencies in the generated implementations and specifications. The situation presented in Section 4.2 provides a concrete example of this.

A common maintenance activity performed by the customer is to augment existing managed objects with additional attributes. This simple extension of the system involves a significant number of changes, not only to the software, but also to the EJB configuration scripts, and the database table definitions. In order to test their changes, the customer augments the generated test code to account for the new attributes. The existing test cases provide a framework within which the customer can add new tests, adding significant value during maintenance. The presence of test cases also provides a well defined way for the customer to certify the system on new platforms.

Domain Logic Test Harnesses. The presence of test-harness scaffolding saves time during development. Developers are able to ignore the details of how the framework alters the signature of the business methods and how the security model must be initialized for testing, thereby reducing the barrier to manual test-code creation.

Also, business rules are pieces of a system that often change after initial development. The presence of existing domain-logic test cases (written by the initial developers) encourages their maintenance in parallel with changes to business rules, increasing the chance that the entire system will be tested as it evolves.

4.2 Development-Time Benefits of Generated Test Cases

Above, we describe the benefits that generated test code provides during development activities and during maintenance activities. In that context, maintenance activities are defined as being those that occur after a generated system is delivered to the customer and, in fact, performed by the customer, not the developer. The bulk of our experience with systems generated by MODEST is with development activities performed during the creation of a few prototype systems.

As the design of the family of systems matures and the artifact template code stabilizes, the subtlety of bugs being found increases. Additionally, as MODEST supports more advanced features, occasionally these features interact in subtle ways when operating in domains with higher complexity.

One such advanced feature is support for cascading deletes in the management EJB layer (see Figure 5). This feature allows a managed-object instance and all other instances reachable from it across explicit relationships to be deleted with a single method call. For example, in the domain specified in Figure 3, a `Car` instance and all of the `Driver` instances associated with it can be deleted by a single cascading delete operation. During an atomic delete operation, logic in the management layer ensures that an object cannot be deleted if it will leave a parent object with an unsatisfied relationship. However, during cascading deletes this check is disabled, since it is known that the parent object

will be deleted immediately after the child object is deleted. One of the prototype systems has a fairly complex domain model, and initially the integration tests that exercise cascading deletes failed. By making incremental changes to the domain specification, regenerating the system, and rerunning the integration tests, developers traced the problem back to the section of the management EJB artifact template that disabled the relationship checks during cascading deletes.

Without the generated test code being a standard part of the system, this bug might not have been found during development. Furthermore, the ability to rapidly regenerate test cases from modified domain specifications provides an invaluable tool to aid in debugging.

4.3 The Cost of Generating Test Code

Above, we present the benefits of test-code generation. However, to developers of generative tools there are costs associated with the generation of test code. The principal cost is the additional effort required of developers to create and maintain the artifact templates that generate test code. In order to evaluate this effort, we have used some simple metrics to measure the size and complexity of artifact templates. The values of these metrics are presented in tables 1 and 2. By examining these metrics we can get a feel for the relative level of effort in artifact template creation and maintenance for the test code compared to the effort needed to create and maintain the templates for application code.

Table 1. Average Relative Complexity of MODEST Test-Code Templates

Type	**Sub-Templates**	**XSL Elements**	**Parent Queries**
Test Code (10)	4	92	1
Application Code (27)	9	184	1
Unit Tests (6)	3	52	2
Integration Tests (3)	6	177	0
Test Harnesses (1)	5	77	0

The three measures of artifact template size and complexity presented in tables 1 and 2 are: (1) the number of sub-templates; (2) the number of XSL elements; and (3) the number of XPath parent queries.

XSL style sheets process XML documents by matching sub-templates against the structure and data contained in the XML document[2]. Sub-templates can also be named and called as functions. The number of sub-templates gives some indication of both the size and complexity of an artifact template, since they

[2] In this paper we refer to the entire XSL style sheet as the template; this corresponds to the top-level `xsl:stylesheet` element. Sub-templates correspond to the `xsl:template` elements that are children of `xsl:stylesheet`.

Table 2. Total Relative Complexity of MODEST Test-Code Templates

Type	Sub-Templates	XSL Elements	Parent Queries
Test Code (10)	42	927	17
Application Code (27)	247	4968	41
Unit Tests (6)	18	317	16
Integration Tests (3)	19	533	1
Test Harnesses (1)	5	77	0

represent the basic data-processing unit of an XSL style sheet. The numbers for these are shown in the second column of tables 1 and 2.

XSL style sheets are themselves XML documents containing special XSL elements intermingled with elements from the output XML document. XSL processors handle the special elements, which contain both control instructions and data-expansion instructions, and pass through the output elements without interpretation. The second metric, XSL element count, gives an indication of the amount of parameterization of the output document, and hence the complexity of the template. The numbers for XSL element counts are shown in the third column of tables 1 and 2.

The third metric, XPath parent queries, is another way to measure the complexity of the XSL template. XPath is a language for querying paths in an XML document. XPath expressions are used extensively in XSL documents for matching templates against parts of the input document, and for selecting parts of the input document to which templates should be applied. Since input documents to an XSL transform are always XML documents, they are inherently hierarchical. In the simplest case the output document has the same hierarchical decomposition as the input document, which means that the transformation can proceed in a top-down fashion. As the output document structure diverges from the input document structure, XPath queries that move up the hierarchy are needed. We refer to these as XPath *parent queries*. They begin with the XPath parent axis shortcut "..". We use a count of these XPath parent queries to represent the level of structural difference between the input and output documents. Keep in mind that when generating Java source code, MODEST employs a two-stage transformation process in an effort to reduce the complexity of artifact templates, principally by removing the need for extensive XPath parent queries.

The first two rows in tables 1 and 2 show the values of our metrics for test code and application code, respectively. These data are intended to provide a feel for the relative complexity of test-code templates compared to application-code templates. The remaining three rows of the tables break down the values for the three different kinds of test code. The number in parenthesis in the first column of both tables is the number of style sheets that were analyzed to come up with the numbers in each row.

Table 1 presents the average metric values. These data represent the relative level of complexity of a single artifact template of each of the representative template types. These data show that, on average, the size and complexity of

test-code templates is less than that of application-code templates. It also shows that out of the three test-code template types, the integration test-case templates require the largest number of XSL elements to generate their desired output.

Table 2 shows the total metric values. This is intended to represent the total effort required to generate templates of the various types. The high-level result shown by these data is that test-code templates are seemingly smaller and simpler than application-code templates. Assuming that our metrics are a reasonable measure of a template's complexity and size, this implies that test-code generation is a fraction of the overall effort required to generate all of a system's code.

5 Related Work

The work described in this paper lies at the confluence of some well-established areas of software engineering research and practice. In this section we briefly review related work in the areas of OMG's Model Driven Architecture, model-based testing, and enterprise Java code generation.

5.1 Model Driven Architecture

At a conceptual level, the design of MODEST shares a number of similarities with the OMG's Model Driven Architecture (MDA) [12]. The basic idea of the MDA is that enterprises can insulate themselves from the volatile nature of the commercial middleware market by focusing their energies on creating Platform Independent Models (PIMs) of their business functions and relying on standard mappings and/or platform experts to map their PIMs into Platform Specific Models (PSMs). The models discussed in the MDA specification are UML models. In the MODEST system, the domain specification is platform independent, but much more restricted than a generic UML model.

Much of the MDA approach is centered around transformations and mappings between UML models at different levels of abstraction, and between PIMs and PSMs. The following mappings have been enumerated.

- PIM to PIM: enhancing, filtering, or specializing models without introducing any platform-dependent details.
- PIM to PSM: projecting a sufficiently refined PIM onto a model of the execution environment.
- PSM to PSM: refining a platform-dependent model.
- PIM to PSM: abstracting existing implementation details into a platform-independent model.

MODEST employs the first three mapping types, which can all be viewed as refinement mappings. The generalization mapping, PIM to PSM, is not utilized in MODEST, since there is a very clear distinction between which artifacts are generated and which must be produced manually. In MODEST, manually generated artifacts often have to conform to interfaces that are generated. However,

there is no mechanism for changes to the structure of manual artifacts to be propagated back up to the higher-level models.

Gervais [6] outlines a methodology that is based on both MDA and the Open Distributed Processing Reference Model (RM-ODP) [1]. Gervais proposes a process for modeling the domain-specific features of a system in a "Behavioral Model" and the high-level technological features of the system in an "Engineering Model". These would be merged into a platform-specific "Operational Model", which could then be used as the basis for generating implementation artifacts. While MODEST's domain specification fills the same role as Gervais' behavioral model, its artifact templates are too low level to even be considered an operational model. Future plans for MODEST include higher-level models for engineering features similar to what Gervais proposes.

5.2 Model-Based Testing

Due to the time-consuming nature of test-case and test-data creation, there have been many studies aimed at generating them automatically. Many approaches focus on the use of high-level formal specifications as the input to their test-generation schemes.

A framework for conducting performance tests of EJB applications is introduced by Liu et al. [10]. Their primary objective is to be able to compare the performance trade offs that are present in different J2EE-compliant servers in the absence of significant application-level logic. Interestingly, the testing they perform is somewhat model driven, since their test-case selection is driven by a common model of the trade offs that are expected to exist within the common feature sets present in J2EE servers.

Grundy, Cai, and Liu [7] discuss the SoftArch/MTE system. This system enables the automatic generation of prototype distributed systems based on high-level architectural objectives. Their emphasis is on performance testing, not on application functionality testing. Many of the generative techniques used in MODEST are similar, in particular the use of XML/XSL to generate code, database schemata, deployment descriptors, and build files.

Dalal et al. [3,4] present a model-based approach to application functionality testing. In their studies, formal specifications of functional interfaces to various telecommunications systems were made available by a software development team. A combinatorial approach was used to generate a set of covering test-data pairs, which was used to find several failures that were not discovered by the existing testing infrastructure.

Mats [11] and Gu and Cheng [8] present two approaches to the derivation of test cases for communications protocols specified using SDL, while Gupta, Cunning, and Rozenbilt [9] discuss an approach to generating test cases for embedded systems. These approaches differ from ours in that they derive test cases for a particular system from formal specifications, as opposed to creating templates that can generate test code for any system instance.

5.3 Enterprise Java Code Generation

Generation of code for Enterprise Java systems is a fairly common activity. A popular approach is to generate the Home and Remote EJB interfaces, deployment descriptors, and stubs for an EJB implementation, all from a simple description of a database table[3]. The majority of these simple code generators are *compositional* generators, in which the descriptor contains the modular decomposition that the generator needs to amplify. The approach used by MODEST is to allow for domain-specific modeling at a higher level than the software components. The modular decomposition is embedded in the generated build script and the XSL style sheets that comprise the bulk of the MODEST system.

6 Conclusion

In this paper we presented our experience in generating test code in parallel with application code, using a model-driven generator of component-based, distributed systems targeted at the EJB framework. Our experience and evaluation suggests that the level of effort required for the test code is only a small fraction of the overall effort, and brings with it significant benefits. We believe that test-code generation should become a common feature of all model-driven generative systems.

While MODEST is a useful tool that achieves the business goals laid out for it, it only raises the level of abstraction a few small steps above the implementation. A future goal for our work is to use higher-level models for both domain and engineering features. Additional work is aimed at utilizing MODEST's generated test code to experiment with design and implementation trade offs, similar to the approach outlined by Grundy, Cai, and Liu [7] for testbed generation, except that our target would be to evaluate full-featured systems.

Acknowledgments

The authors would like to thank Dan Weiler, President of Chronos Software, Inc., and co-developer of MODEST.

This work was supported in part by the Defense Advanced Research Projects Agency under agreement number F30602-00-2-0608. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

[3] EJEN, `http://ejen.sourceforge.net`
Jenerator, `http://www.visioncodified.com`

References

1. ISO IS 10746-x. ODP reference model part x. Technical report, International Standards Organization, 1995.
2. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
3. S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, and C. M. Lott. Model-based testing of a highly programmable system. In *Proc. 9th International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998.
4. S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C. M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE '99)*, 1999.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
6. M.P. Gervais. Towards an MDA-oriented methodology. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002)*, 2002.
7. J. Grundy, Y. Cai, and A. Liu. Generation of distributed system test-beds from high-level software architecture descriptions. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001.
8. Z.Y. Gu and K.E. Cheng. The derivation of test cases from sdl specifications. In *Proceedings of the 30th Annual Southeast Regional Conference*. ACM Press, 1992.
9. P. Gupta, S.J. Cunning, and J.W. Rozenbilt. Synthesis of high-level requirements models for automatic test generation. In *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*, 2001.
10. Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen. Designing a test suite for empirically-based middleware performance prediction. In *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, 2002.
11. L. Mats. Selection criteria for automated TTCN test case generation from SDL. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998.
12. J. Miller and J. Mukerji. Model driven architecture (MDA). *OMG Document* `ormsc/2001-07-01`, July 2001.