Interface-driven Model-based Generation of Java Test Drivers

Mark Blackburn, Robert Busser, Aaron Nauman, T-VEC Technologies/SPC
Ramaswamy Chandramouli, National Institute of Standards and Technology

This paper extends prior work in model-based verification and describes interface-driven analysis that combines with a requirement model to support automated generation of Java test scripts. It describes concepts of models and test driver mappings using examples for testing security functionality of an Oracle database using Java and Structured Query Language(SQL) test drivers. Although the modeling and testing are focused on database security capabilities, the described concepts can be applied to develop test drivers for many other products.

Keywords: Test Automation Technology and Experience, Interface-driven Model-Based Test Automation, Java and SQL Test Driver Generation, Security Testing, Database Testing

1 Introduction

The combination of model-based verification and test automation has helped reduce cost, provide early identification of requirement defects, and improve test coverage [RR00; KSSB01; BBNKK01; BBN01d; Sta00; Sta01]. This paper extends prior work in model-based verification and recommends the use of interface-driven analysis with requirement modeling to support automated test generation. The interface analysis provides key information that results in **test driver mappings** that specify the relationships between model variables and the interfaces of the system under test. The paper describes the concepts of models and test driver mappings using examples for testing security functionality of an Oracle database using Java and Structured Query Language (SQL) test drivers[1]. Recommendations are provided for performing the modeling of textual requirements in conjunction with interface analysis to support reuse of models and their associated test driver mappings. These recommendations were derived while extending an early experimental model of one small set of requirements to several other groups of interrelated requirements. The resulting insights have been applied to other industry applications and are useful for understanding how to scale models and the associated test driver mappings to support industry-sized verification projects.

1.1 Background

The National Institute of Standards and Technology (NIST) [Cha99] assessed the feasibility of automating security functional testing using the Test Automation Framework[2] (TAF). TAF

[1] One of the key requirements for the environment required the testing to be executed against the Oracle database engine through a Java/JDBC connection.

[2] Certain Commercial Products are mentioned in this paper. This does not imply recommendation or endorsement by National Institute of Standards and Technology nor does it imply that the products mentioned are necessarily the best available for the purpose.

integrates various government and commercially available model development and test generation tools to support defect prevention and automated testing of systems and software. T-VEC [BBNC01] is a component tool in TAF.T-VEC supports test vector generation, test driver generation, requirement test coverage analysis, and test results checking and reporting [BB96]. **Test vectors** include inputs as well as the expected outputs with requirement-to-test traceability information. The test driver mappings and the test vectors are inputs to the test driver generation, which produces test drivers that are executed against the implemented system.

Although the modeling and testing examples are focused on security functionality of a database, the results and recommended approaches are general for testing most applications. TAF has been applied to other applications in various domains including critical applications for aerospace (Mars Polar Lander) [BBNKK01], medical devices, flight navigation, guidance, autopilots, display systems, flight management and control laws, engine controls, and airborne traffic and collision avoidance. TAF has also been applied to non-critical applications like workstation-based Java applications with GUI user interfaces, databases, client-server, web-based, automotive, and telecommunication applications. The related test driver generation has been developed for many languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL, etc.) as well as proprietary languages, COTS test injection products (e.g., DynaComm®, WinRunner®) and test environments. Most users of the approach have reduced their verification/test effort by 50 percent [KSSB01, Saf00].

1.2 Contributions

This paper provides pragmatic guidance for combining interface analysis and requirement modeling. These recommendations for defining interfaces that provide better support for testability are valid for all forms of testing. Although this paper describes why interface-driven modeling has benefits for testing a released product, it has been applied during development with many additional benefits, which are described in Section 2.3.

1.3 Organization of Paper

Section 2 provides an overview of the method and tools, while providing concept definitions, guidance on interface definitions and analysis, and organizational roles and best practices. Section **Error! Reference source not found.** discusses the security requirements and database interface details using examples. Section 4 discusses the test driver mapping and associated Java support required for test driver generation. Section 5 provides conclusions concerning the use of Java for automated test driver support and summarizes the benefits of interface-driven model-based testing.

1.4 Related Work

Formal Models of software functions have been developed for supporting software engineering functions like design, coding and testing. Some examples of modeling approaches can be found in [HJL96; PM91; Sch90], with examples that support automated test generation [BBN01a; BBN01b; BBN01c; BBNC01, BBNKK01]. Asisi provides a historical perspective on test vector generation and describes some of the leading commercial tools [Asi02]. Pretschner and Lotzbeyer briefly discuss Extreme Modeling that includes model-based test generation [PL01], which is similar to uses of TAF as discussed in Section 2.3. There are various approaches to model-based

testing and Robinson hosts a website that provides useful links to authors, tools and papers [Rob00].

2 Method and Tool Overview

The TAF support, as shown in Figure 1, involves three main roles of development, including Requirement Engineer, Design/Implementation Engineer, and Test Engineer. A requirements engineer performs requirement analysis and typically documents the requirements in text. A designer/implementer develops the technical solution, which includes system/software architecture, design and implementation. Test engineers clarify the requirements in the form of a **verification model**, which specifies behavioral requirements in terms of the interfaces for the system under test.[3] This is in contrast to a "pure" requirement model, which specifies the requirements in terms of *logical entities* representing the environment of the system under test [PM91; Sch90; HJL96]. Verification modeling from the interfaces is analogous to the way a test engineer develops tests in terms of the specific interfaces of the system under test. TAF translators convert verification models into a form where the T-VEC system generates test vectors and test drivers, with requirement-to-test traceability information that allows failures to be traced backwards to the requirement.

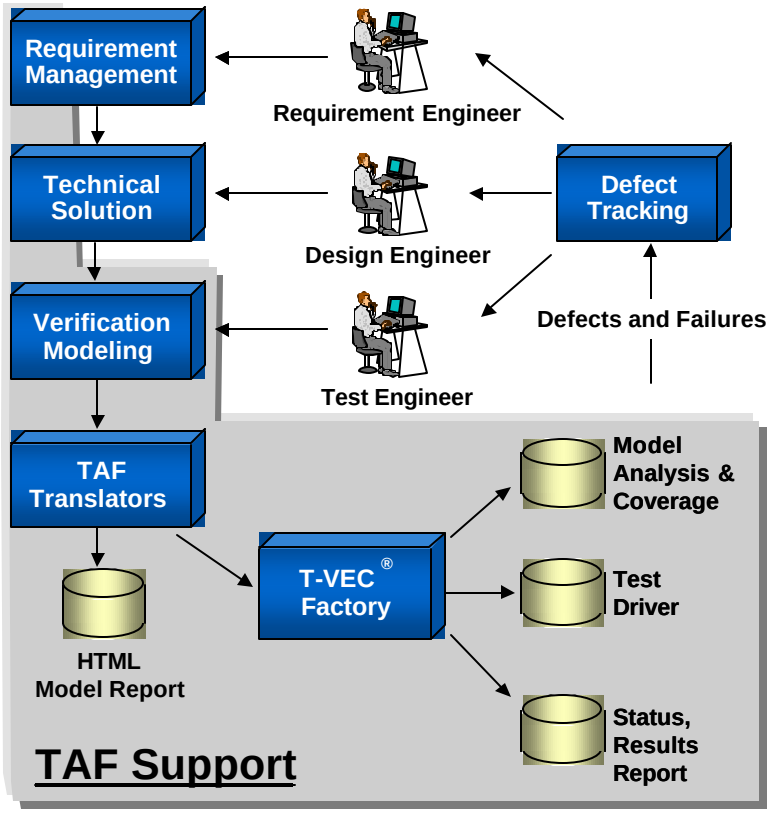


Figure 1. Test Automation Framework Life Cycle Automation

[3] A design engineer typically defines the interfaces, and component interfaces are typically documented in a application programming interface (API) or other interface documents.

2.1 Verification Modeling Process

Figure 2 provides a detailed perspective of the verification modeling process flow. A test engineer is supplied with various inputs. Although it is common to start the process with poorly defined requirements, inputs to the process can include requirement specifications, user documentation, interface control documents, application program interface (API) documents, previous designs, and old test scripts. A verification model is composed of a model and one or more test driver mappings. A test driver mapping consists of object mappings and a schema (pattern). Object mappings relate the model objects to the interfaces of the system under test. The schema defines the algorithmic pattern to carry out the execution of the test cases. The one selected for use in this paper is a Java test driver schema tailored to interface with an Oracle database through a Java Database Connectivity (JDBC) application programming interface (API).

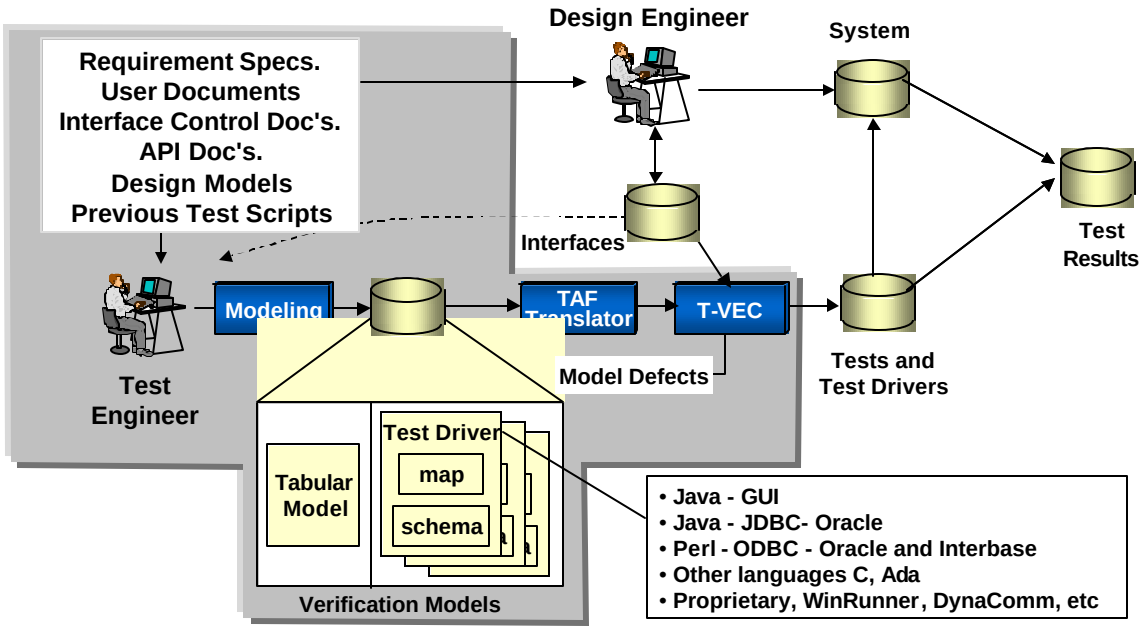


Figure 2. Verification Model Details

Models are typically developed incrementally. The models are translated and T-VEC generates test vectors. T-VEC also detects untestable requirements (i.e., requirements with contradictions). Test drivers are produced from the test vectors using the test driver mappings and schema information. Details are provided in Section 4.

2.2 Nature of the Requirements Model

The modeling language used for developing the requirements model in our approach is the Software Cost Reduction (SCR) method [HJL96]. SCR is a table-based requirements model that has been very effective and relatively easy to learn for test engineers [KSSB01]. Although design engineers commonly develop models based on state machines or other notations like the Unified Modeling Language (UML), users and project leaders observed that test engineers find it easier to develop requirements for test in the form of tables (See [BBN01a] for details). The modeling

notations supported by tools for the SCR method have well-defined syntax and semantics allowing for a precise and analyzable definition of the required behavior.

2.3 Why Interface-Driven Modeling?

It may seem appropriate to first develop models from the requirements, but when developing models for the purpose of testing, the models should be developed in conjunction with analysis of the interfaces to the component or system under test. Modeling the behavioral requirements is usually straightforward and easier to evolve once the interfaces and operations are understood because the behavioral requirements, usually defined in text, must be modeled in terms of variables that represent objects accessible through interfaces.

2.2.1 Modeling Perspectives

Models are described using specification languages, usually supported through graphical modeling environments. Specification languages provide abstract descriptions of system and software requirement and design information. Cooke et al. developed a hierarchical scheme that classified specification language characteristics [CGDDTK96]. Independent of any specification language, Figure 3 illustrates three categories of specifications based on the purpose of the specification. Cooke et al. indicates that most specification languages usually are based on a hybrid approach that integrates different classes of specifications.

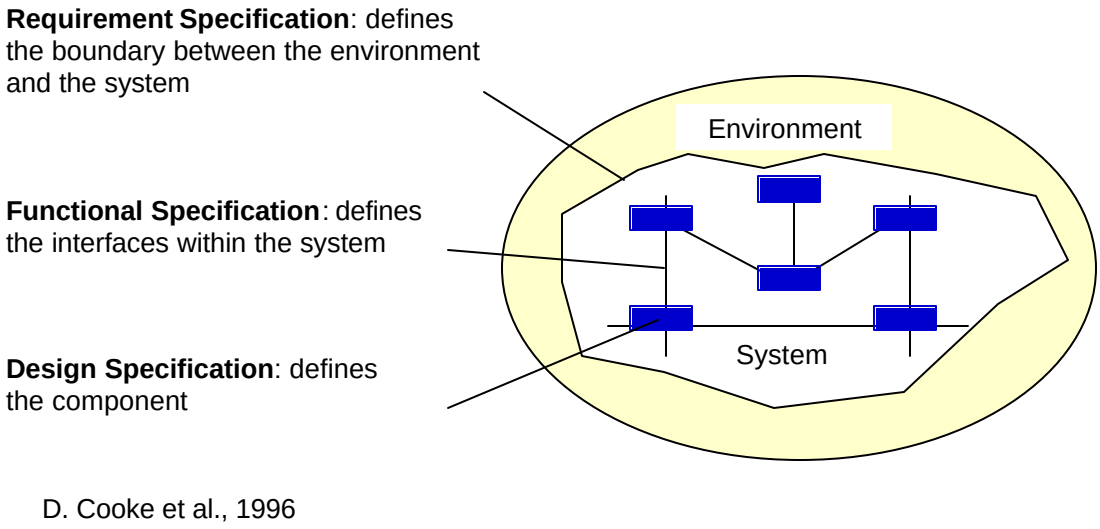


Figure 3. Specification Purposes

Requirement specifications define the boundaries between the environment and the system and, as a result, impose constraints on the system. Functional specifications define behavior in terms of the interfaces between components, and design specifies the component itself. A specification may include behavioral, structural, and qualitative properties. Behavioral properties define the relationships between inputs and outputs of the system [Sim69]; structural properties provide the basis for the composition of the system components; and qualitative requirements [YZCG84] define nonfunctional requirements. Often, modeling languages support certain elements of both

requirement and functional specifications and collectively these two types of specifications are called functional requirements [Rom85].

A verification model, in the context of this paper, is best classified as a functional specification. The requirements are defined in terms of the interfaces of the components. The term interface is used loosely in this paper. An **interface** is a component's inputs and outputs, along with the mechanism to set inputs, including state and history information, and retrieve the resulting outputs. Some components or systems may require sequences of function calls to initialize a component or system, as well as additional calls to place the system in a particular state prior to setting the inputs for testing.

2.2.2 Database Interfaces

For database security requirements the interfaces include the data dictionary (sometimes referred to as system tables) that hold security information and reflect the results of security operations. For each set of modeled requirements it is important to determine the data dictionary views and the SQL commands associated with the requirements, and determine how those database tables are modified to reflect the "correct" or "incorrect" results. Once the interfaces and the SQL operations that affect those tables are understood, it's usually easy to develop the test driver mappings and models hand-in-hand.

2.2.3 Interface Accessibility

Since the interfaces are the key to developing executable test drivers, it is necessary to understand the interfaces of the system under test even prior to the development of the requirements model. This will ensure that the assumed inputs and outputs that are being modeled can be realized through the interfaces. If the interfaces are not formalized or completely understood, requirement models can still be developed, but developing the object mappings (which map model entities to interface components) can become a complex process. In addition, if the component interfaces are coupled to other components, the components are typically not completely controllable through separate interfaces. This too can complicate the modeling and testing process. Consider the following conceptual representation of the set of components and interfaces shown in Figure 4.

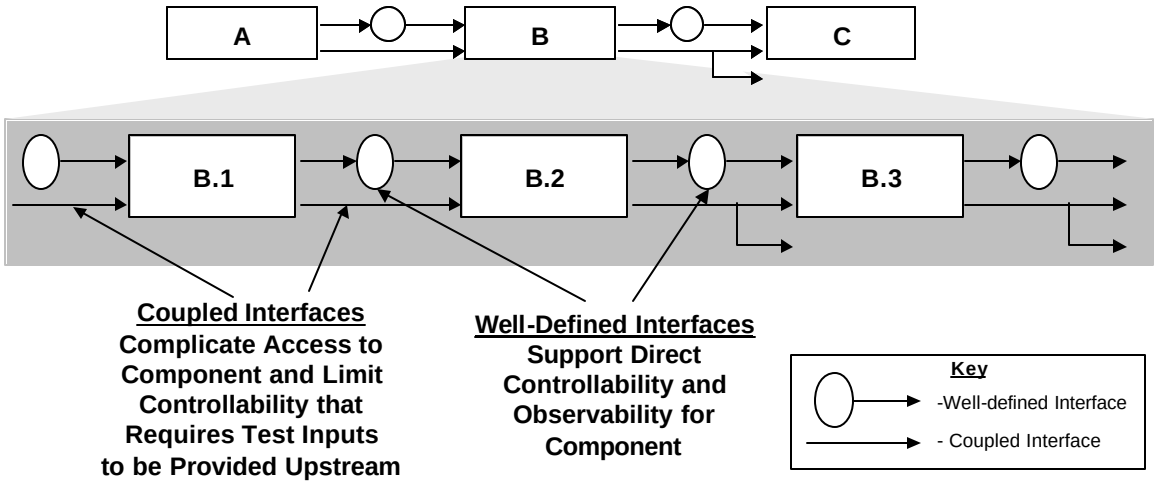


Figure 4. Conceptual Components of System

To support a systematic verification approach that can be performed in stages where each component is completely verified with respect to the requirements allocated to it, the interfaces to the component should be explicitly and completely accessible, either using global memory, or better through get and set methods/procedures as reflected in Figure 4. For example, if the inputs to the B.2 component of higher-level component B are completely available for setting the inputs to B.2, and the outputs from the B.2 functions can be completely observed, then the functionality within B.2 can be completely specified and systematically verified. However, if interfaces from other components, such as B.1 are not accessible, then some of the functionality of the B.2 component is coupled with B.1, and the interfaces to B.2, must also include interfaces to B.1, or to other upstream components, such as component A. This interface coupling makes the test driver interfaces more complex to describe, but also forces the behavioral modeling to be described in terms of functionality allocated to combinations of components. The coupling reduces the reuse of components, and increases the regression testing effort due to the coupled aspects of the system components. The problems associated with testing highly coupled systems can be problematic for model-based testing, but also negatively impacts any type of testing. As discussed in Section 2.3, we have observed that interface-driven modeling has helped foster better system design by reducing the coupling, but also helps provide better support for testing.

Systematic test coverage can typically be achieved directly from the verification model if the components of the system can be tested individually. Component integration testing can later be performed from higher-level models to ensure that the integration of the components (i.e., the contractual obligation of the integration) is systematically and completely verified.

2.3 Organizational Best Practices

Interface-driven modeling can be applied after development is complete as is the case for security testing of an Oracle database. However, significant benefits have been realized when it was applied during development. Ideally, test engineers work in parallel with developers to stabilize interfaces, refine requirements, and build models to support iterative test and development. Test engineers write the requirements for the products (which in some cases are very poorly documented) in the form of models, as opposed to hundreds or thousands of lines of test scripts.

They generate the tests vectors and test drivers automatically. During iterative development, if the component behavior, the interface, or the requirements change, the models are modified, and test cases and test drivers are regenerated, and re-executed. The key advantages are that testing proceeds in parallel to development. Users like Lockheed Martin state that test is being reduced by about fifty percent or more, while describing how early requirement analysis significantly reduces rework through elimination of requirement defects (i.e., contradiction, inconsistencies, feature interaction problems) [Saf00, KSSB01]. This typical and pragmatic use of TAF parallels eXtreme Programming (XP) [Bec99] where tests are created before the program. However, others refer to this model-based method as Extreme Modeling (XM) [PL01; BBWL00], which applies the principles to write tests prior to coding. With XP test code is developed manually, but with XM the requirements are modeled and the tests are generated.

3 Developing the Verification Model for Oracle Database

As already stated, a verification model consists of a requirement (behavioral) model defined in terms of interface components. In our case study, the requirements model for Oracle 8.0.5 product was developed using the security functional requirements and associated function definitions found in the Oracle's Common Criteria Security Target document [Ora00]. The interfaces for Oracle DBMS product consists of JDBC commands, SQL commands and the Oracle data dictionary views. The JDBC commands were obtained from JDBC commands in the Sun's Web site, data dictionary views from Oracle8 Reference document, and the SQL commands were obtained from the Oracle8 SQL Reference document, the last two being supplied with the Oracle software.

Prior efforts focused on developing verification models for the security functionality, referred to as "Granting Object Privilege Capability (GOP)" [BBNC01]. While extending the model to support Identification & Authentication, Security Management, and Session Management, we observed that it reduces work when "low-level" primitive models and their associated test driver mappings are developed first so that the low-level models and test driver mappings can be reused as primitives in higher-level requirement models. Developing from the lowest-level interfaces is not an absolute requirement, but if this approach is applied to a larger verification effort, the resulting verification model leverages reusable model elements that are directly related to reusable test driver interface mappings.

3.1 Security Requirement Interfaces Analysis

Prior to, or in conjunction with, modeling the requirements, the database interfaces associated with the requirements are analyzed to identify common tables, SQL commands, and common test driver mappings that can be extended and maintained as the product evolves. Model variables are used to represent database tables, objects, privileges and relationships. Consider the example of Granting Object Privilege. The requirements state:

```
A normal user (the grantor) can grant an object privilege to another user, role
or PUBLIC (the grantee) only if:
  a) the grantor is the owner of the object; or
  b) the grantor has been granted the object privilege with the GRANT OPTION.
```

The SQL operations that are directly related to the granting of the object privileges include:

```
GRANT <privilege> ON <object> TO <user | role | PUBLIC> [WITH GRANT OPTION]

Where <privilege> can be: ALTER, EXECUTE, INDEX, INSERT, READ, REFERENCES,
SELECT, UPDATE, ALL, and the GRANT OPTION is optional.

And, where <object> is a database schema object like a table, view, sequence,
procedure, function, package, or snapshots.

And, where <user> is a database user, <role> is a defined database role, and
<PUBLIC> represents all users.
```

However, there are some initial privileges and dependent SQL commands that are related to the GRANT SQL command. These involve the creation of a user, role, or session.

- When a user is created with the CREATE USER command, the user's privilege is empty.
- To log on to Oracle, a user must have CREATE SESSION system privilege. After creating a user, the user must be granted this privilege.

There are numerous other cases where additional constraints restrict grant privileges on various object types. These details are beyond the scope of this paper, and are not discussed.

The data dictionary table that is affected, or can be used to determine if a particular GRANT operation is successful, is DBA_TAB_PRIVS. The data dictionary view (based on this table) lists all grants on objects in the database. It has attributes that indicate, the GRANTEE (user to whom access was granted), object owner, name of the object, GRANTOR (user who performed the grant operation), privilege, and an indication of whether the privilege can be propagated by the grantee to another user.

3.2 Security Models and Interface Specifications

As shown in Figure 5, the behavioral requirements are derived from the requirement text in the Oracle Security Target, like Grant Object Privilege. The requirements are defined in terms of the model variables that represent the interface defined in terms of the data dictionary and SQL commands. The interfaces are declared as model variables using the modeling tool. The mapping for the model variable defines how to affect that variable within the test execution environment. For example, a GRANT SQL command must be issued to affect an object's privilege.

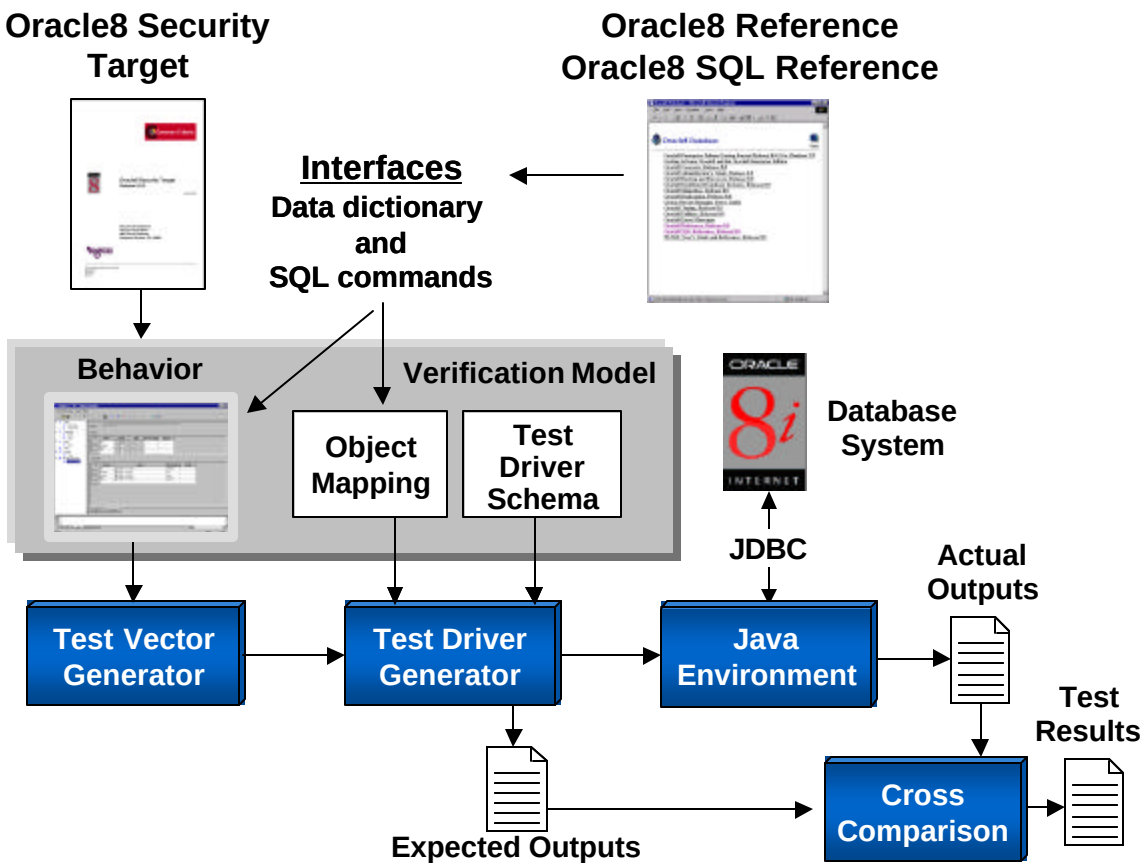


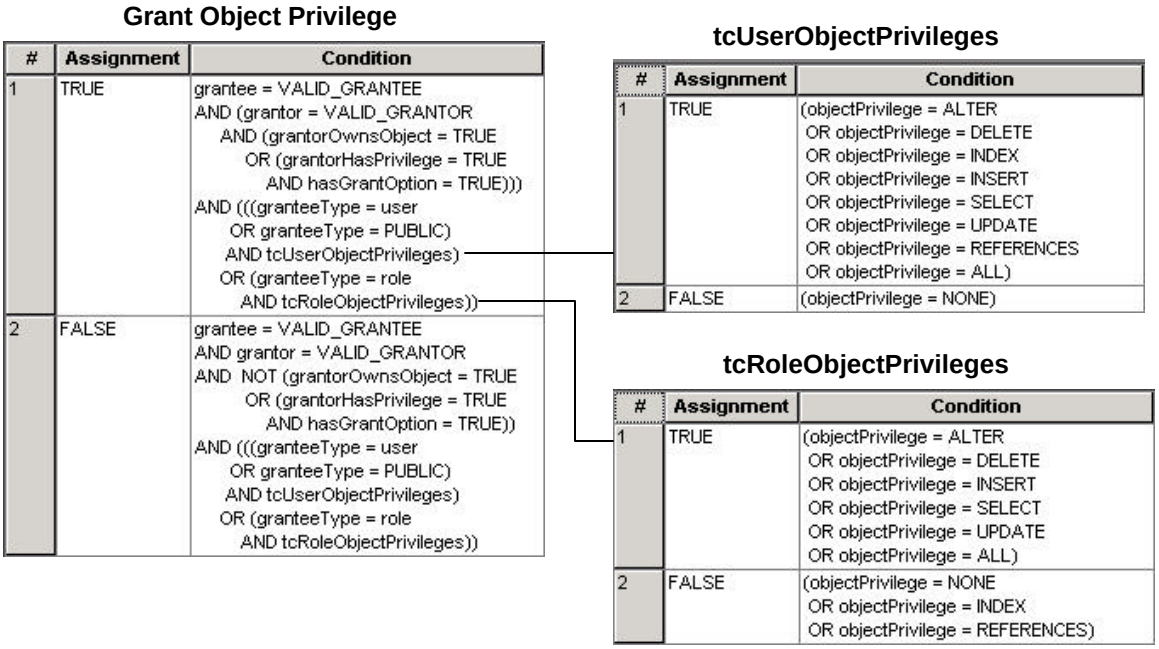
Figure 5. Detailed Process Flow

As shown in Figure 5, the model is input to the test vector generator, and the resulting test vectors are combined with the object mappings and test driver schema (details provided in Section 4) to produce a Java test driver. The executing test driver communicates with the Oracle database through a JDBC connection to carry out the tests. The actual outputs for each test are captured by the test driver during test execution and stored for post processing. Finally a cross comparison tool compares the expected outputs against the actual outputs and produces a test results log that indicates the pass/fail status for each test vector.

3.2.1 Modeling Security Properties

Each security property is modeled as a Boolean object in a manner similar to Grant Object Privileges as shown in Figure 6. The conditions associated with the TRUE output or the positive sense for the model is the valid set of conditions required for Granting Object Privilege. Each test case for the TRUE case should result in valid actions with respect to the security relationships established for that case. The FALSE cases are negative conditions, which establish realistic database relationship, but the corresponding test attempts to execute invalid operations, from a security perspective that should be denied as an invalid security response. Some operations cause failures because the database responds with an error message when improper or unauthorized actions are requested. This general approach is used to model each security requirement to ensure that proper security exists for authorized actions, while unauthorized actions are not permitted.

Row 1 of the model for Grant Object Privilege, shown in Figure 6, with the assignment TRUE describes the conditions in which the grant object privilege should be permitted. When the grantee and the grantor are valid database users, then an object privilege should be granted if the grantor owns the object, or if the grantor has been granted object privileges with the GRANT OPTION. In addition, the model defines additional conditions where the grantee (reflected by granteeType) can be a user, PUBLIC or role. The term variable tcUserObjectPrivileges references another condition table that enumerates the set of objectPrivileges (e.g., ALTER, DELETE, INDEX, INSERT, etc.) that are valid, and should be tested. If the granteeType is a role, then the term tcRoleObjectPrivileges defines a subset of the valid ObjectPrivileges that apply to roles.



Grant Object Privilege

#	Assignment	Condition
1	TRUE	grantee = VALID_GRANTEE AND (grantor = VALID_GRANTOR AND (grantorOwnsObject = TRUE OR (grantorHasPrivilege = TRUE AND hasGrantOption = TRUE))) AND (((granteeType = user OR granteeType = PUBLIC) AND tcUserObjectPrivileges) OR (granteeType = role AND tcRoleObjectPrivileges))
2	FALSE	grantee = VALID_GRANTEE AND grantor = VALID_GRANTOR AND NOT (grantorOwnsObject = TRUE OR (grantorHasPrivilege = TRUE AND hasGrantOption = TRUE)) AND (((granteeType = user OR granteeType = PUBLIC) AND tcUserObjectPrivileges) OR (granteeType = role AND tcRoleObjectPrivileges))

tcUserObjectPrivileges

#	Assignment	Condition
1	TRUE	(objectPrivilege = ALTER OR objectPrivilege = DELETE OR objectPrivilege = INDEX OR objectPrivilege = INSERT OR objectPrivilege = SELECT OR objectPrivilege = UPDATE OR objectPrivilege = REFERENCES OR objectPrivilege = ALL)
2	FALSE	(objectPrivilege = NONE)

tcRoleObjectPrivileges

#	Assignment	Condition
1	TRUE	(objectPrivilege = ALTER OR objectPrivilege = DELETE OR objectPrivilege = INSERT OR objectPrivilege = SELECT OR objectPrivilege = UPDATE OR objectPrivilege = ALL)
2	FALSE	(objectPrivilege = NONE OR objectPrivilege = INDEX OR objectPrivilege = REFERENCES)

Figure 6. Example Model for Grant Object Privilege

3.3 Relationship of Security Requirements and Interfaces

Table 1 provides a summary for several modeled requirements. Each row provides a brief summary of a requirement, the related data dictionary views, associated SQL command that are primarily used to affect the operation, and related commands that are referred to as dependent commands.

For example, the Grant Role Privilege command, like the Grant Object Privilege command describes the requirements for granting and revoking role privileges. The primary data dictionary table from which the results of the granted role privilege can be retrieved is the DBA_ROLE_PRIVS (database administrator role privileges). The SQL commands that are used to grant/revoke privileges are GRANT and REVOKE, and the related SQL commands include CREATE, INSERT, SELECT and others. The operations and test driver commands required to support Grant Role Privilege overlap Grant Object Privilege. More importantly, much of the functionality for other requirements like DISABLE and ENABLE roles subsume many of the tested requirements developed for GRANT and REVOKE roles.

Table 1. Detailed Security Specification Analysis

Requirement Summary	Data Dictionary Items	SQL Command	Dependant Command
Disable roles	DBA_ROLE_PRIVS SESSION_ROLES	SET ROLE	GRANT, ALTER,
Enable roles	DBA_ROLE_PRIVS SESSION_ROLES	SET ROLE	GRANT
Grant object privileges	DBA_TAB_PRIVS	GRANT	CREATE, INSERT, SELECT
Grant/revoke role privileges	DBA_ROLE_PRIVS	GRANT/ REVOKE	CREATE, INSERT, SELECT
Grant system privileges	DBA_SYS_PRIVS	GRANT	CREATE, INSERT, SELECT
Revoke privileges	DBA_TAB_PRIVS	REVOKE	GRANT
Every object uniquely identified, even if deleted	ALL_OBJECTS		CREATE, INSERT, SELECT, DELETE

4 Test Driver Generation

This section provides a brief summary of test driver generation. The details of the models, test vectors and test drivers are beyond the scope of this paper. In addition, to understand the test driver support requires some understanding of Java, SQL and operational details of an Oracle database. Additional details including the security requirement models (in HTML), test vectors, object mappings, test driver schema, test drivers and instructions for installing and executing the test drivers against an Oracle database are available for download from: http://www.software.org/pub/taf/Reports.html.

4.1 Creation of Test Oracle Database

The test driver dynamically creates and deletes database information in the form of users, roles, database tables and values. Although most manual database-related testing is performed using populated databases, model-based test generation systematically populates the database with test data derived from the model. This allows automated test execution without manual assistance. The models are constructed in a way that is independent of any specific populated database. There are some specific database conditions that must be established prior to the execution of the tests. For example a database administrator must install the database and the Oracle database test execution requires the "TEMPORARY" tablespace to be available during execution.

4.2 Test Driver Application Programming Interface and Language

The test driver API discussed in this paper is based on JDBC API using Java that makes SQL calls to the database. In prior work Perl test drivers used an Object Database Connectivity (ODBC) API to inject SQL calls to both Oracle and Interbase databases. Although each language provides suitable support for performing the test execution, we believe that there is more effort involved in developing the Java/JDBC support as opposed to the Perl/ODBC support for test driver generation.

4.3 Java Test Driver Support

The test driver generation support capabilities are provided by a Java infrastructure to:

- Retrieve global test configuration settings that can be configured to direct the test driver mechanisms to use user-specified options such as log directory, output file directory, system user and password, etc.
- Retrieve test vector parameters during test execution
- Log test operation
- Create test output file
- Establish an Oracle database connection and SQL execution through JDBC
- Specify an interface to which each test must conform along with helper methods
- Provide global constants
- Provide a framework for test execution

4.3.1 Test Driver Packaging

The test driver support is packaged using a Java package com.tvec.support, which contains the following classes:

- ConfigManager – provides access to the global test configuration settings
- Constants – set of constants used by the tests
- Context – used to retrieve and set test vector parameters
- Logger – provides classes to write log files and output files
- SQLUtils – provides database access
- TestImpl – abstract class with the test interface and helper methods
- TestRunner – framework for running classes that implement TestImpl

4.3.2 Operational Scenario

The TestRunner class contains the entry point for running tests that implement TestImpl. Executing TestRunner performs as follows:

1. Read the global configuration file to determine the log file directory, the output file directory, and the maximum number of users
2. Initialize the test database, which deletes existing test table space, create a new test table space
3. Get the test vectors from TestImpl by calling TestImpl.getTestVectors. For each test vector:
    1. Create default data based on the user-specified number of standard users
    2. Call TestImpl.setupTest to setup the test environment further
    3. Call TestImpl.runTest to perform the test and return a Boolean result
    4. Write the result of the test to the output file
    5. Call TestImpl.cleanupTest to do standard cleanup needed to restore test environment.
    6. Perform cleanup of standard users, tables, roles, and profiles

4. Exit.

4.3.3 TestImpl Interface

The TestImpl class contains four methods that must be implemented when creating a test, including: setupTest, runTest, cleanupTest, and getTestVectors**.**

- setupTest performs additional database configuration beyond the creation of the standard users
- runTest performs test execution
- cleanupTest restores the database to a known state to support the next test vector
- getTestVectors retrieves the inputs for the current test.

4.3.4 SQLUtils

The SQLUtils class handles the database connectivity and SQL execution. It maintains a user-authenticated connection that is used to execute SQL commands. The connection is only lost when a disconnection-related operation or another connect call is performed.

5 Summary

This paper provides pragmatic guidance for combining interface analysis and requirement modeling to support model-based test automation. The model-based testing method and tools described in this paper have been demonstrated to significantly reduce cost and effort for performing testing, while being demonstrated to identify requirement defects that reduce costly rework. These recommendations for defining interfaces that provide better support for testability are valid for all forms of testing. Although this paper describes why interface-driven modeling has benefits for testing a released product, it has been applied during development with many additional benefits. Organizations see the benefits of using interface driven model-based testing to help stabilize the interfaces of the system early, while identifying common test driver support capabilities that can be constructed once and reused across related tests. In addition, parallel development of verification modeling is beneficial in development and helps identify requirement defects early to reduce rework. This concept has been characterized as eXtreme modeling, which is similar to eXtreme programming.

Although this paper discusses modeling and test automation for security requirements, the tools and method are generally applicable because they have been used in several other application domains. Finally, this paper discusses the use of Java test drivers, but in prior work Perl test drivers were developed for both Oracle and Interbase databases. Although each language provides suitable support for performing the test execution, we believe that there is more effort involved in developing the Java support as opposed to the Perl support for test driver generation.

6 References

[Asi02] Aissi, S.,Test Vector Generation: Current Status and Future Trends, Software Quality Professional, Volume 4, Issue 2, March 2002.

[Bec99] Beck, K., Extreme Programming Explained: Embrace Change. Addison Wesley, 1999.

[BBN01a] Blackburn, M.R., R.D. Busser, A.M. Nauman, Removing Requirement Defects and Automating Test, STAREAST, May 2001.

[BBN01b] Blackburn, M. R., R.D. Busser, A.M. Nauman, How To Develop Models For Requirement Analysis And Test Automation, Software Technology Conference, May 2001.

[BBN01c] Blackburn, M. R., R.D. Busser, A.M. Nauman, Eliminating Requirement Defects and Automating Test, Test Computer Software Conference, June 2001.

[BBNC01] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Chandramouli, Model-based Approach to Security Test Automation, In *Proceeding of Quality Week 2001*, June 2001.

[BBNKK01] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Knickerbocker, R. Kasuda, Mars Polar Lander Fault Identification Using Model-based Testing, Proceeding in IEEE/NASA 26th Software Engineering Workshop, November 2001.

[BBN01d] Busser, R. D., M. R. Blackburn, A. M. Nauman, Automated Model Analysis and Test Generation for Flight Guidance Mode Logic, Digital Avionics System Conference, 2001.

[BBWL00] Boger, M., T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. In Proc. Extreme Programming and Flexible Processes in SW Engineering (XP'00), 2000.

[CGDDTK96] Cooke, D., A. Gates, E. Demirors, O.Demirors, M. Tankik, B. Kramer, Languages for the Specification of Software, Journal of Systems Software, 32:269-308, 1996.

[Cha99] Chandramouli R., Methodology for Automated Security Testing", NIST Request for Proposal, Nov 1999.

[HJL96] Heitmeyer, C., R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications. *ACM TOSEM*, 5(3):231-261, 1996.

[KSSB01] Kelly, V. E.L.Safford, M. Siok, M. Blackburn, Requirements Testability and Test Automation, Lockheed Martin Joint Symposium, June 2001.

[PL01] Pretschner, A., H. Lotzbeyer, Model Based Testing with Constraint Logic Programming: First Results and Challenges, Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto, May 2001.

[PM91] Parnas, D., J. Madley, Functional Decomposition for Computer Systems Engineering (Version 2), TR CRL 237, Telecommunication Research Inst. of Ontario, McMaster University, 1991.

[Rob00] Robinson, H., http://www.model-based-testing.org/.

[Rom85] Roman, G.C., A Taxonomy of Current Issues in Requirements Engineering, IEEE Computer, 18(4):14-23, 1985.

[RR00] Rosario, S., H. Robinson, Applying Models in Your Testing Process, Information and Software Technology, Volume 42, Issue 12, 1 September 2000.

[Sch90] van Schouwen, A.J., The A-7 Requirements Model: Re-Examination for Real-Time System and an Application for Monitoring Systems. TR 90-276, Queen's University, Kinston, Ontario, 1990.

[Sta00] Statezni, David. Test Automation Framework, State-based and Signal Flow Examples, *Twelfth Annual Software Technology Conference*, April 30 - May 5, 2000.

[Sta01] Statezni, David. T-VEC's Test Vector Generation System, *Software Testing & Quality Engineering*, May/June 2001.

[Saf00] Safford, Ed L. Test Automation Framework, State-based and Signal Flow Examples, *Twelfth Annual Software Technology Conference*, April 30 - May 5, 2000.

[YZCG84] Yeh, R.T., P. Zave, A.P. Conn, G.E. Cole, Software Requirements: New Directions and Perspectives, in Handbook of Software Engineering, Editors C. R. Vick and C. V. Ramamoorthy), Van Nostrand Reinhold, 1984.