Model-based Approach to Security Test Automation

Mark Blackburn
Robert Busser
Aaron Nauman
Software Productivity Consortium/T-VEC
2214 Rock Hill Road, Herndon, VA 20170

Ramaswamy Chandramouli
National Institute of Standards and Technology
Computer Security Division
100 Bureau Drive, Mail Stop 8930
Gaithersburg, MD 20899-8930

Abstract

Security functional testing is a costly activity typically performed by security evaluation laboratories. These laboratories have struggled to keep pace with increasing demand to test numerous product variations. This paper summarizes the results of applying a model-based approach to automate security functional testing. The approach involves developing models of security function specifications (SFS) as the basis for automatic test vector and test driver generation. In the application, security properties were modeled and the resulting tests were executed against Oracle and Interbase database engines through a fully automated process. The findings indicate the approach, proven successful in a variety of other application domains, provides a promising approach to security functional testing.

1. Introduction

Software security is a software quality issue that continues to grow in importance as software systems manage continually increasing amounts of critical corporate and personal information. The use of the Internet to manage and exchange this data has heightened the need for secure software architectures, especially Internet-based architectures. At the same time, shortened development and deployment cycles for software make it difficult to conduct adequate security functional testing to verify whether software systems exhibit the expected security behavior.

Presently, developing and executing security functional tests is time-consuming and costly. Security evaluation laboratories are struggling to meet demands to test many product variations produced in short release cycles. As a result, the National Institute of Standards and Technology (NIST) initiated a program to develop methods and tools for automating security functional testing [1]. Security Functional Testing helps to verify whether the behavior of a product or system conforms to the security features claimed by the manufacturer.

NIST and its sponsors initiated a multi-phase investigation to assess the use of a model-based approach to automate security functional testing. Several model-based approaches were assessed as part of the investigation. The approach in this paper was selected because it provided end-to-end support including model development, model analysis, automated test generation, automated test execution in multiple environments, and results analysis. The assessment of this approach has demonstrated the feasibility of modeling security function specifications (SFS) to automate testing for various products and target platforms. NIST believes this should improve the economics of security functional testing for security evaluation laboratories, as well as commercial organizations that perform security testing.

1.1 Background

The core capabilities underlying this approach were developed in the late 1980s and proven through use in support of FAA certifications for flight critical avionics systems [2]. The approach supports requirement-based test coverage mandated by the FAA with significant life cycle cost savings [3; 4]. The approach reduces cost, effort, and cycle-time by eliminating requirement defects and automating testing [5]. Safford's presentation summarized the benefits:

- Better quality requirements for design and implementation help eliminate rework in those phases as well as during test
- Verification modeling can reduce the time normally spent in verification test planning by up to 50 percent
- Test generation from a verification model can eliminate up to 90 percent of the manual test creation and debugging effort
- Both the number of test cases and the phasing of their execution can be optimized, eliminating test redundancy
- A known level of requirements coverage can be planned, and measured during test execution

The approach and tools described in this paper have been used for modeling and testing system, software integration, software unit, and hardware/software integration functionality. It has been applied to critical applications like cardiac rhythm management, flight navigation, guidance, autopilot logic, display systems, flight management and control laws, airborne traffic and collision avoidance. The approach supports

automated test driver generation in a variety of open languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL), as well as, proprietary languages, and test environments.

2. NIST Requirements For Automated Security Functional Testing

NIST wishes to develop a methodology and supporting toolkit to automate Security Functional Testing. This automation will help security evaluation laboratories meet the demand for product testing. The automation approach is based on expressing a product's security functional requirements in a model and using the supporting toolkit to automatically generate tests needed to verify security properties. A model of system security properties is referred to as a **Security Verification Model**. The supporting toolkit processes these models to:

- Check the SFS for contradictions, feature interaction problems, and circular definitions. This analysis ensures the underlying SFS are consistent as a basis for testing.
- Generate test cases from the SFS expressed in the models. These test cases must effectively demonstrate that an implementation satisfies the SFS. Ideally, test cases include test inputs, expected behavior or outputs, and an association between each test and the specification from which it was derived. Test cases of this form are referred to as test vectors to distinguish them from tests cases that include only test inputs.
- Check SFS-to-test traceability and report whether each specification has an associated test.

As a single fault in security functionality can annul the entire system's security behavior, it is critical that the model representation of the SFS be complete. The techniques for developing tests to verify the security properties must also provide 100 percent test coverage of the security properties. As system security behavior is often a product of both trusted and untrusted system components, complete testing minimizes the risk of using untrusted components in a system. This risk minimization is an additional objective of the NIST effort.

3. Methodology and Toolkit for Automating Security Functional Testing

The methodology and toolkit described in this paper are based on a model-based test automation approach referred to as the Test Automation Framework (TAF). The TAF integrates various modeling tools, like the SCRtool[1] for modeling system and software requirements with the test automation tool T-VEC.[2] The TAF approach was tailored to automate security functional testing through security verification models for two distinct environments, as shown in Figure 1. The resulting process includes:

- Model security function specifications in SCR specifications using the SCRtool
- Translate SCR specifications into T-VEC test specifications using an existing SCR-to-T-VEC model translator [6; 7]
- Generate test vectors from the transformed SCR specification
- Develop test driver schemas and object mappings for various target test environments
- Generate Perl test drivers for an SQL database using an ODBC database interface
- Generate Java test drivers for an SQL database using a JDBC database interface

The process for the automated security testing approach begins with the development of a model for the ISO/IEC 15408 Security Target[3] specification for Oracle 8 Database Server using the SCRtool. An SCR-to-T-VEC translator, developed by the Software Productivity Consortium and T-VEC, translates the SCR model to a T-VEC test specification. T-VEC tools automatically generate test vectors. The T-VEC test driver generator produces test drivers to execute tests against an Interbase 6.0 database server and an Oracle 8.0.5 database server. The test drivers are executed and the results are compared with the expected results from the test vectors to determine each product's compliance to the security properties.

The primary effort to support security functional testing involves modeling security properties with the SCRtool and developing test driver schemas to automate execution of SQL statements for the two different test environments.

---

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

[2] The Software Productivity Consortium develops TAF translators and methods. The Software Cost Reduction (SCR) method and associated modeling tool, SCRtool, were developed by the Naval Research Laboratory [9]. The T-VEC Test Vector Generation System is commercially available from T-VEC Technologies, Inc.

[3] An ISO/IEC 15408 Security Target is a document that contains a set of Security Functional Requirements, corresponding implementation features and a set of Security Assurance Requirements for an IT product or system, written in a format that corresponds to an international standard.

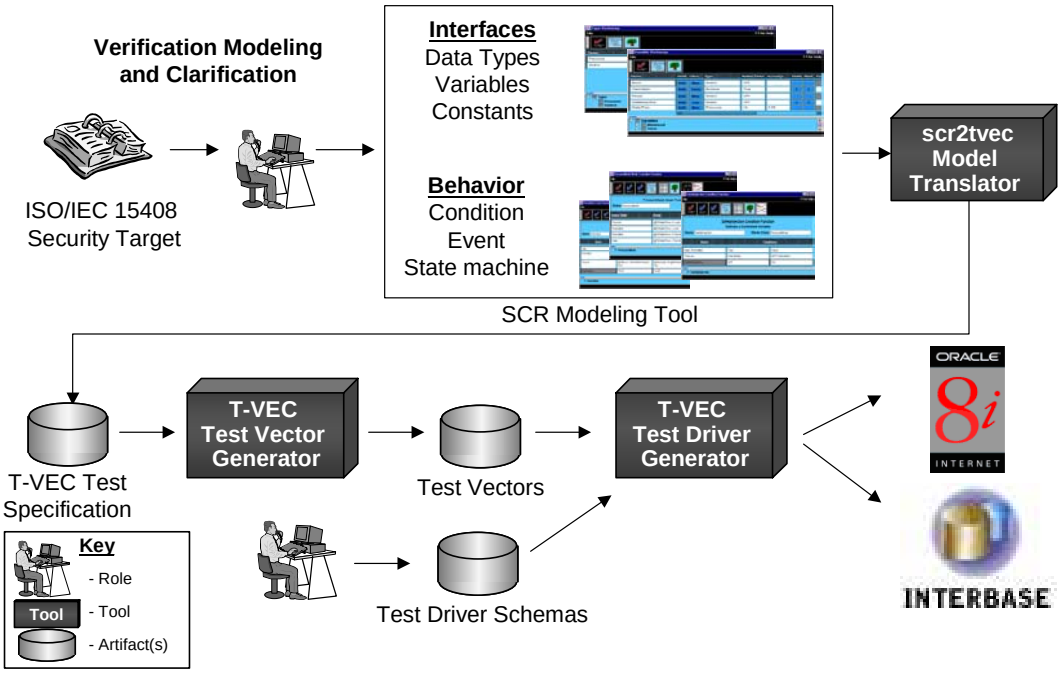

Figure 1. Process Flow Through the Tools

4. Security Verification Model

This section describes the development of a security verification model using the SCRtool. First, basic SCR modeling concepts are described. This is followed by a description of a security functional specification that is then refined into a verification model.

4.1 SCR Modeling Concepts

SCR is a table-based modeling approach that models system and software requirements. SCR represents system inputs as **monitored variables**, system outputs as **controlled variables** and intermediate values as **term variables**. Variables are defined as primitive types (e.g., Integers, Float, Boolean, Enumeration) or as user-defined types. Behavior is defined using a tabular approach relating four model elements: modes, conditions, events, and terms. A **mode class** is a state machine, where system states are called system modes and the transitions of the state machine are characterized by guarded events. A **condition** is a **predicate** characterizing a system state. An **event** occurs when any system entity changes value. Terms and controlled variables are functions of input variables, modes, or other terms. Their values are defined in the model through event or condition tables.

4.2 Security Function Specifications

The SFS used in the assessment is defined in the Oracle8 Security Target document [8]. This document describes the security functionality (behavior) claimed by Oracle and is submitted along with the product for security evaluation. A subset of the security functionality, referred to as "Granting Object Privilege Capability (GOP)," was modeled. The following sections describe the process of modeling the GOP specification, which is a part of the *Granting and Revoking Privileges and Roles* functionality. The GOP is defined in the Oracle8 Security Target as:

Granting Object Privilege Capability (GOP) - 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.

A role represents a group of related users. The keyword PUBLIC represents all users.

4.3 Analysis of Security Function Specification

Developing SCR models requires identifying the system monitored (input) and controlled (output) variables, and defining the relationships between them. The value of each output is defined in terms of the system inputs. Term variables are introduced to decompose the relationships, which can be reused within the model. Breaking the GOP specification into clauses supports identifying variables and relationships. Table 1 contains the variables and relationships associated with each clause of the GOP specification.

Table 1. Variables and Terms

Specification Statement/Clause	Variables	Terms
A normal user (the grantor) can grant an object privilege to another user, role or PUBLIC (the grantee)	grantor, grantee, selectedObj, selectedObjPriv, granteeType	
GOP (a) – a grantor can grant an object privilege to a grantee if the grantor owns the object	grantor, grantee, selectedObj, selectedObjPriv, selectedObjOwner	grantor_owns_object
GOP (b) – a grantor (that does not own the object) can grant object privileges to the grantee if he/she possess the object privilege with GRANT OPTION.	grantor, grantee, selectedObj, selectedObjPriv, grantedObj, grantedObjPriv	has_grantable_obj_privs

From the analysis above, the monitored (input) variables identified in the system can be refined into the following set:

- grantor – user granting an object privilege
- grantee – user being granted an object privilege
- selectedObj – object selected for a particular grant operation
- selectedObjPriv – type of privilege (object access mode) (ALL, SELECT, INSERT, UPDATE, DELETE, etc) on the object that is selected for a particular grant operation
- selectedObjOwner – the owner of the object selected for a particular grant operation
- granteeType – type of grantee for a particular grant operation as defined in the first sentence of the GOP textual specification; grantee is a user, role, or PUBLIC
- grantedObj – object for which the grantor holds grantable privileges
- grantedObjPriv – the privilege associated with the grantedObj (above) the grantor holds

Two other variables are related to the concept of a role; a role is a type of grantee as defined in the first sentence of the GOP textual requirement. The related variables include:

- valid_roleID – identifier for a valid role
- granteeRoleID – role identifier of the grantee if the granteeType is "role"

There can be one or more roles defined and known by the database system. The variable valid_roleID is used to refer to a specific role known within the system and used in various test cases.

The GOP specifications stipulate the restrictions governing the granting of an object privilege by one user to another. An SCR model of this specification should ensure that when all model conditions are satisfied, the privilege granting operation is successful. This output is modeled as the Boolean controlled variable:

- grant_obj_priv_OK – the object privilege granting operation executes successfully (TRUE) or fails (FALSE)

4.4 Modeling Security Function Specifications

Once the system's data is defined, its behavior can be modeled. In SCR, this involves defining the values of the controlled (output) variables through condition, event, or mode tables. These tables define the value of a variable in terms of monitored (input) variables, term (intermediate) variables, and mode (state) machines. Figure 2 provides a representation for the GOP specification (stated in section 4.2). The condition table for the output grant_obj_priv_OK references two terms. The specification for GOP(a) is associated with the term grantor_owns_object and the specification for GOP(b) is associated with the term has_grantable_obj_privs.

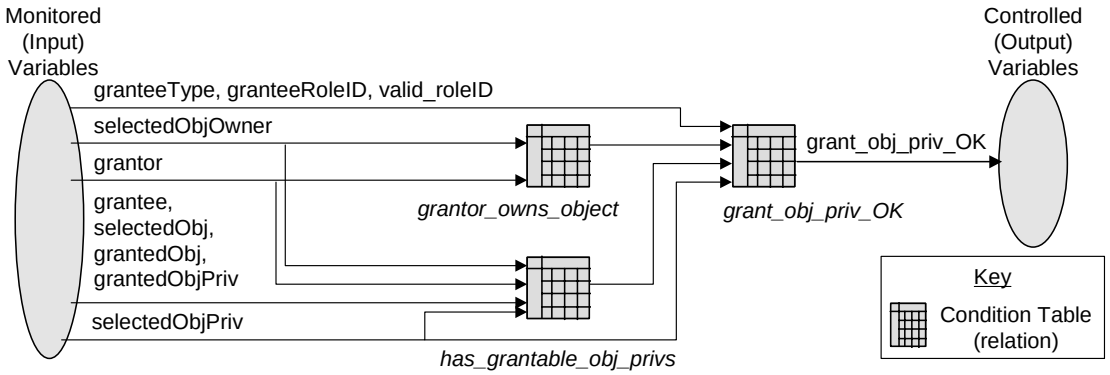

Figure 2. Model Structure for "Granting Object Privilege" Capability

A value of a **term variable** is defined through a condition or event table as an intermediate value. Terms can be referenced as part of the constraints or value calculations of other terms or controlled variables. They reduce the complexity of the model by simplifying expressions and eliminating redundancies. The following sections describe the terms used in defining the value of grant_obj_priv_OK. The model details are described in the following sections, and Figure 3 provides the detailed tabular specification for the term and condition variables.

Modeling the Term grantor_owns_object. The term grantor_owns_object defines the conditions when the grantor owns the object for which he/she is granting a privilege associated with the object to another user, role, or PUBLIC (grantee). It specifies that the term grantor_owns_object is TRUE when the condition grantor = selectedObjOwner is satisfied, otherwise, the term is FALSE.

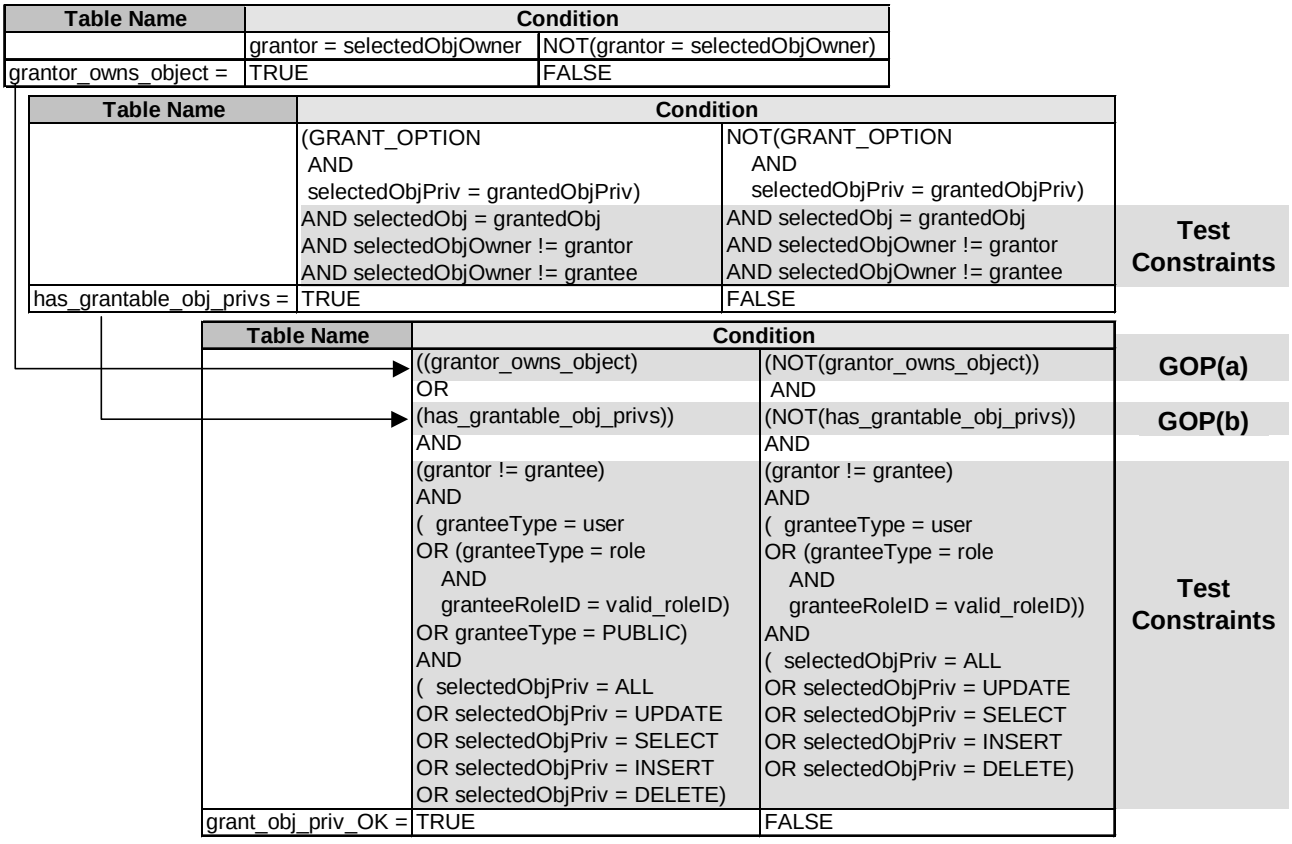

Table Name	Condition	
	grantor = selectedObjOwner	NOT(grantor = selectedObjOwner)
grantor_owns_object =	TRUE	FALSE

Table Name	Condition		
	(GRANT_OPTION AND selectedObjPriv = grantedObjPriv)	NOT(GRANT_OPTION AND selectedObjPriv = grantedObjPriv)	
	AND selectedObj = grantedObj AND selectedObjOwner != grantor AND selectedObjOwner != grantee	AND selectedObj = grantedObj AND selectedObjOwner != grantor AND selectedObjOwner != grantee	**Test Constraints**
has_grantable_obj_privs =	TRUE	FALSE	

Table Name	Condition		
	((grantor_owns_object)	(NOT(grantor_owns_object))	**GOP(a)**
	OR	AND	
	(has_grantable_obj_privs))	(NOT(has_grantable_obj_privs))	**GOP(b)**
	AND	AND	
	(grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID) OR granteeType = PUBLIC) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)	(grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID)) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)	**Test Constraints**
grant_obj_priv_OK =	TRUE	FALSE	

Figure 3. Behavioral Specifications for "Granting Object Privilege" Capability

Modeling the Term has_grantable_obj_privs. The GOP(b) specification states that if a user wishes to grant an object privilege to another user, role, or PUBLIC and does not own the object, the user must have been granted that object privilege on that object with the GRANT OPTION. The term has_grantable_obj_privs defines these conditions. The term is TRUE when:

1. The grantor holds the privilege with the ability to grant to others (GRANT_OPTION is TRUE)
2. The privilege on the selected object the grantor is granting to others is the same as he/she already holds (i.e., the selectedObjPriv is the grantedObjPriv)
3. Test Constraints – additional constraints that ensure conditions labeled 1 and 2 above exercise realistic scenarios. The conditions ensure the following:
   - The selected object is the object for which the grantor holds the privilege (i.e., the selectedObj is the grantedObj).
   - The owner of the object is not the grantor
   - The owner of the object is not the grantee

The term is FALSE when primary conditions, labeled 1 and 2 above, are negated while the test constraints are applied to test all combinations.

Modeling the Output grant_obj_priv_OK. The definition of grant_obj_priv_OK completes the model of the GOP specification. Its definition includes references to the

terms defined previously, as well as additional constraints on monitored variables. The two potential values for grant_obj_priv_OK include:

- grant_obj_priv_OK = TRUE - test case conditions are such that the privilege can be granted
- grant_obj_priv_OK = FALSE - test case conditions are such that the privilege cannot be granted

The conditions are divided into three groups to support explanation. The groups include:

1. GOP(a) – grantor can grant privilege to a grantee because the grantor owns the object
2. GOP(b) – grantor can grant privilege to a grantee because the grantor has been granted object privileges with GRANT OPTION
3. Test Constraints – additional constraints that ensure that the GOP(a) and GOP(b) conditions are fully exercised during test generation. The conditions ensure the following:
   - grantor is not the grantee
   - Each granteeType (user, role, or PUBLIC) is tested. When the granteeType is a role, the granteeRoleID must be a valid role id.
   - Each object privilege type (ALL, UPDATE, SELECT, etc.) is tested.

5. Test Vector Generation

The SCR-to-T-VEC model translator transforms each SCR table into a T-VEC subsystem. The T-VEC compiler converts each subsystem into a set of primitive test specifications that are used for test vector generation [6]. The translated and compiled version of the grant_obj_priv_OK requirement includes 40 test specification paths (TSP). The test vector generator attempts to determine two test vectors for each test specification based on a test selection strategy derived from **domain testing theory**[4]. Table 2 shows a tabular representation of a subset of the 80 test vectors produced for grant_obj_priv_OK. The test vectors include 11 monitored variables and 2 term variables (not shown in the table). The test values shown in Table 2 reflect how the test generator systematically selects low-bound and high-bound test points at the domain boundaries. The input variable ranges and constraints (e.g., relational operators) of the specification define the domain boundaries. For example, vector # 1, grantor has id = 1, grantee has id = 2, is based on low-bound values of the data type range of userIDType, while vector # 2, grantor has id = 4, grantee has id = 3, is based on the high-bound for the data type range. In addition, the test generator creates a test for each value of selectedObj and granteeType.

[4] White and Cohen [10] proposed **domain testing theory** as a strategy for selecting test points to reveal domain errors. It is based on the premise that if there is no coincidental correctness, then test cases that localize the boundaries of domains with arbitrarily high precision are sufficient to test all the points in the domain.

6. Test Driver Generation and Execution

The last step in the automated process involves transforming the tests derived from one common model of the security functional specifications into test drivers that are executed against the product, which in this case includes InterBase 6.0 and the Oracle DBMS product Version 8.05.

The test driver generator combines test driver schemas, user-defined object mappings and test vectors to produce test drivers as illustrated in Figure 4. The test driver schema encodes an algorithmic pattern for test execution for the specific test environment. The object mappings relate model variables to the implementation interface. The test driver generator creates test drivers by repeating the execution steps defined in the schema for each test vector. There are typically four primary steps for executing each test case:

- Set the value of the test output to some value other than what is expected to ensure the output is set appropriately
- Set the test input values
- Cause execution of the test
- Retrieve the output and save the results of the test execution

Test driver schemas describe how to accomplish these steps for a specific test environment using a small language that accesses information about the specification model, data objects, types, ranges, test values, and user customizable information. A schema also describes the form of expected outputs to support results analysis.

Two different test driver schemas and object mapping descriptions were used with the grant_obj_priv_OK model to test two different test environments. The Interbase test driver was developed in Perl using ODBC interface to issue SQL commands. The Oracle test driver was developed in both Perl and Java. The Java test drivers used JDBC to communicate to the database.

The test drivers produced from a common model executed without failure in the Oracle database. The execution of test drivers against the Interbase database did result in test failures, but the failures were associated with restrictions on granting roles. These restrictions are described in the Interbase documentation. In addition, Interbase does not permit users to be created through SQL. These differences where addressed in the test driver schema for the two test environments.

Table 2. Test Vectors for grant_obj_priv_OK

#	TSP	grant_obj _priv_OK	grantor	grantee	grantee Type	grantee RoleID	valid_roleID	selected ObjPriv	objOwner	GRANT_ OPTION	granted ObjPriv	selected Obj	granted Obj
1	1	TRUE	1	2	user	2	2	ALL	1	TRUE	ALL	4	4
2	1	TRUE	4	3	user	1	1	ALL	4	FALSE	SELECT	1	1
3	2	TRUE	1	2	user	2	2	UPDATE	1	TRUE	ALL	4	4
4	2	TRUE	4	3	user	1	1	UPDATE	4	FALSE	SELECT	1	1
5	3	TRUE	1	2	user	2	2	SELECT	1	TRUE	ALL	4	4
6	3	TRUE	4	3	user	1	1	SELECT	4	FALSE	SELECT	1	1
7	4	TRUE	1	2	user	2	2	INSERT	1	TRUE	ALL	4	4
8	4	TRUE	4	3	user	1	1	INSERT	4	FALSE	SELECT	1	1
9	5	TRUE	1	2	user	2	2	DELETE	1	TRUE	ALL	4	4
10	5	TRUE	4	3	user	1	1	DELETE	4	FALSE	SELECT	1	1
▪ ▪ ▪													
77	39	FALSE	1	2	role	1	1	INSERT	3	FALSE	ALL	1	1
78	39	FALSE	4	3	role	2	2	INSERT	2	FALSE	SELECT	4	4
79	40	FALSE	1	2	role	1	1	DELETE	3	FALSE	ALL	1	1
80	40	FALSE	4	3	role	2	2	DELETE	2	FALSE	SELECT	4	4

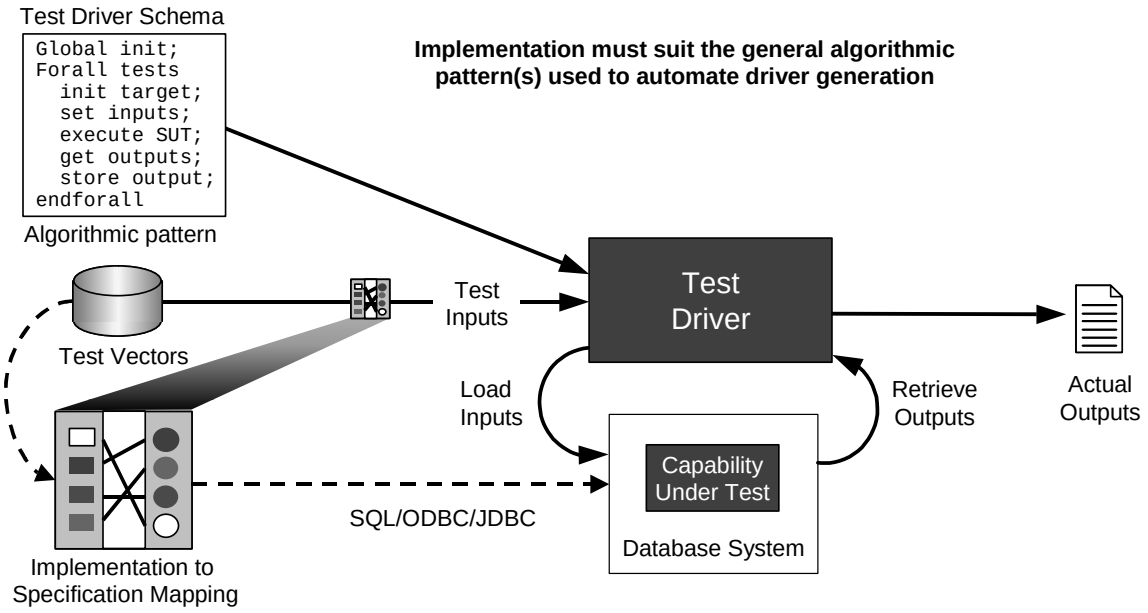

Figure 4. Elements of a Test Driver

7. Summary and Future Work

The TAF approach, customized with specific guidelines for modeling security properties and developing test drivers for databases, satisfies NIST's requirements for an automated model-based approach to automated Security Functional Testing. In the assessment of the approach, security functionality claimed in an Oracle8 Security Target was modeled using the SCRtool. These models were then used as the basis for automated test vector and test driver generation with the T-VEC toolset for two product applications and test environments. This approach reduces the time and effort associated with security testing, while increasing the level of test coverage. These results demonstrate the feasibility of using model-based test automation to improve the economics of security functional testing. Specifically, the potential beneficiaries of the TAF approach are security evaluation laboratories and other commercial organizations that need a cost-effective approach for performing security functional testing.

8. References

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

[2] Blackburn, M.R., R.D. Busser, T-VEC: A Tool for Developing Critical System. In Proceeding of the Eleventh International Conference on Computer Assurance, June, 1996.

[3] Statezni, David, Industrial Application of Model-Based Testing, 16th International Conference and Exposition on Testing Computer Software, June 1999.

[4] Statezni, David. Test Automation Framework, State-based and Signal Flow Examples, Twelfth Annual Software Technology Conference, May 2000.

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

[6] Blackburn, M.R., R.D. Busser, J.S. Fontaine, Automatic Generation of Test Vectors for SCR-Style Specifications, In Proceeding of the 12th Annual Conference on Computer Assurance, June, 1997.

[7] Blackburn, M. R., Using Models For Test Generation And Analysis, Digital Avionics System Conference, October, 1998.

[8] Oracle Corporation, Oracle8 Security Target Release 8.0.5, April 2000.

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

[10] White, L.J., E.I. Cohen, A Domain Strategy for Computer Program Testing. IEEE Transactions on Software Engineering, 6(3):247-257, May, 1980.