Model-based Approach to Security Test Automation

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

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 cost-effective solution to security functional testing.

1 Introduction

Software security is a software quality issue that continues to grow in importance as software systems are used to manage continually increasing amounts of critical corporate and personal information. The use of the Internet to manage and exchange this data on a daily basis has heightened the need for software architectures, especially internet-based architectures, which are secure. At the same time, the 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. The situation calls for improving the economics of security functional testing. As a result, the National Institute of Standards and Technology (NIST) initiated a program to develop methods and tools for automating security functional testing [Cha99]. **Security Functional Testing** verifies whether the behavior of a product or system conforms to the security features claimed by the manufacturer (i.e., the product does what it is supposed to do).

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 accessed as part of the investigation. The approach described in this paper succeeded where others failed to provide 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 Organization of Paper

Section 2 details NIST's vision for a methodology and toolkit to support automated security functional testing. Section 3 provides an overview of a methodology and toolkit that have been effective in satisfying NIST's objectives and that form the basis of this report. Section 4 uses an example to illustrate the development of Security Verification Models to support test automation. Section 5 summarizes the activity of model analysis and test vector generation. Section 6 briefly discusses aspects of test driver generation and test execution.

2 NIST Requirements For Automated Security Functional Testing

NIST wishes to develop a methodology and a supporting toolkit to automate the process of 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 security function specifications (SFS) for contradictions, feature interaction problems, and circular definitions. This analysis ensures that the underlying security function specifications (SFS) are consistent and reasonable as a basis for testing.
- Generate test cases from the security function specifications (SFS) expressed in the models. These test cases must be effective in demonstrating an implementation satisfies the SFS. Ideally, the test cases should 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 generated tests cases that include only test inputs.
- Check for 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 basis for the methodology and toolkit described in this paper is a model-based test automation approach used successfully in various application domains since1996. The approach is referred to as the Test Automation Framework (TAF). The TAF integrates various modeling tools, like the SCRtool[!] for modeling system and software requirements with the test automation tool T-VEC.[*] In this work, that TAF approach was tailored to automate security functional testing through

[!] 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 not does it imply that the products and standards mentioned are necessarily the best available for the purpose

[*] 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 [HJL96]. The T-VEC Test Vector Generation System is commercially available from T-VEC Technologies, Inc.

Security Verification Models. The result is a set of guidelines for modeling security function specifications. The assessment was based on modeling security function specifications in order to automate testing in two distinct environments, as shown in Figure 1. The specific activities carried out in the assessment include:

- Model security function specifications in SCR specifications using the SCRtool
- Translate SCR specifications into T-VEC test specification using an existing SCR-to-T-VEC model translator [BBF97; Bla98]
- Generate test vectors from the transformed SCR specification
- Develop test driver schemas 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

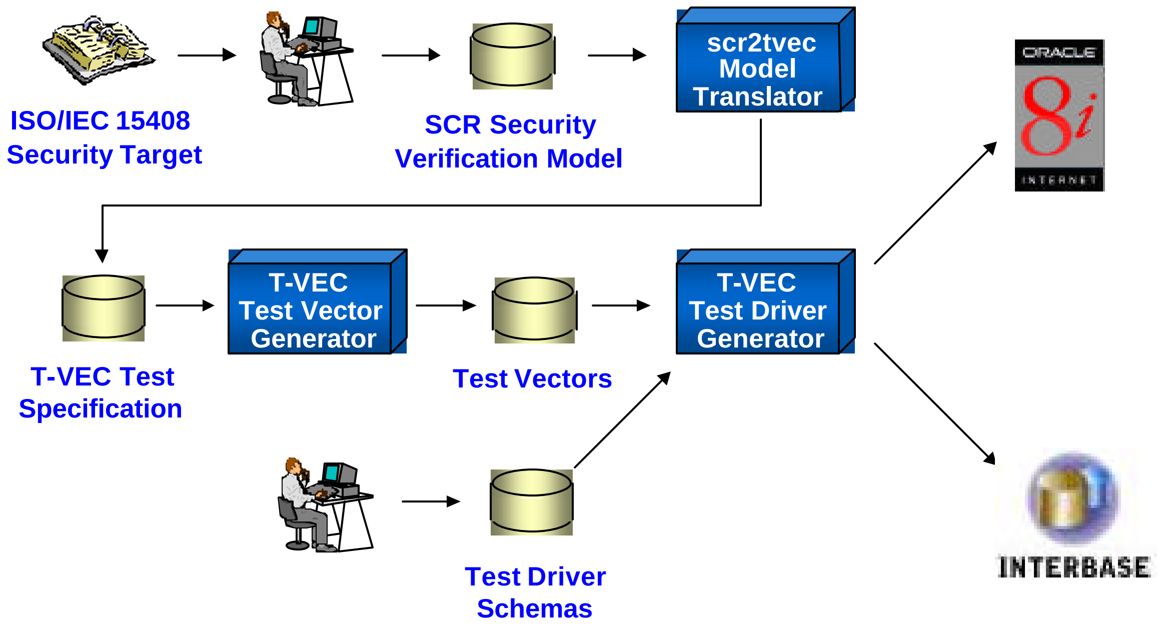

ISO/IEC 15408 Security Target
SCR Security Verification Model
scr2tvec Model Translator
ORACLE 8i INTERNET
T-VEC Test Specification
T-VEC Test Vector Generator
Test Vectors
T-VEC Test Driver Generator
Test Driver Schemas
INTERBASE

Figure 1. Process Flow Through the Tools

Figure 1 illustrates the process for automated security functional testing used in the assessment. First, security properties from ISO/IEC 15408 Security Target[⊥] specification for Oracle 8 Database Server were modeled in SCR with the SCRtool. An SCR-to-T-VEC translator, developed by the Software Productivity Consortium and T-VEC, was used to translate the SCR model to a T-VEC test specification. T-VEC tools were then used on the T-VEC representation of the security properties to automatically generate test vectors (i.e., test cases with test input values, expected output values and traceability information) and specification-to-test coverage metrics. The T-VEC test driver generator was used in the assessment to automatically generate test drivers to execute tests against an Interbase 6.0 database server and an Oracle 8.0.5 database server. These

[⊥] 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.

tests were executed and the results were compared with the expected results from the test vectors to determine each product's compliance to the security properties.[1]

The primary effort in customizing the TAF approach to support security functional testing involved developing heuristics for modeling security properties with SCR and finding techniques for developing test driver schemas to automate execution of SQL statements.

4 Security Verification Model

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

4.1 SCR Modeling Concepts

SCR is a table-based modeling approach, as shown in Figure 2 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 security function specifications (SFS) used in the assessment are defined in the Oracle8 Security Target document [Ora00]. 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", was modeled in the assessment. The test vectors derived from the model were used to generate test drivers for two different database servers, Interbase 6.0 and the Oracle 8.0.5.

The following sections describe the process of modeling the *Granting Object Privilege Capability (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.

[1] The process of SCR model translation, test vector generation, test driver generation, and execution against the Interbase database using Perl and ODBC completed in 2 minutes and 54 second running on a 400 MHz Windows NT machine with 256 KB of memory.

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

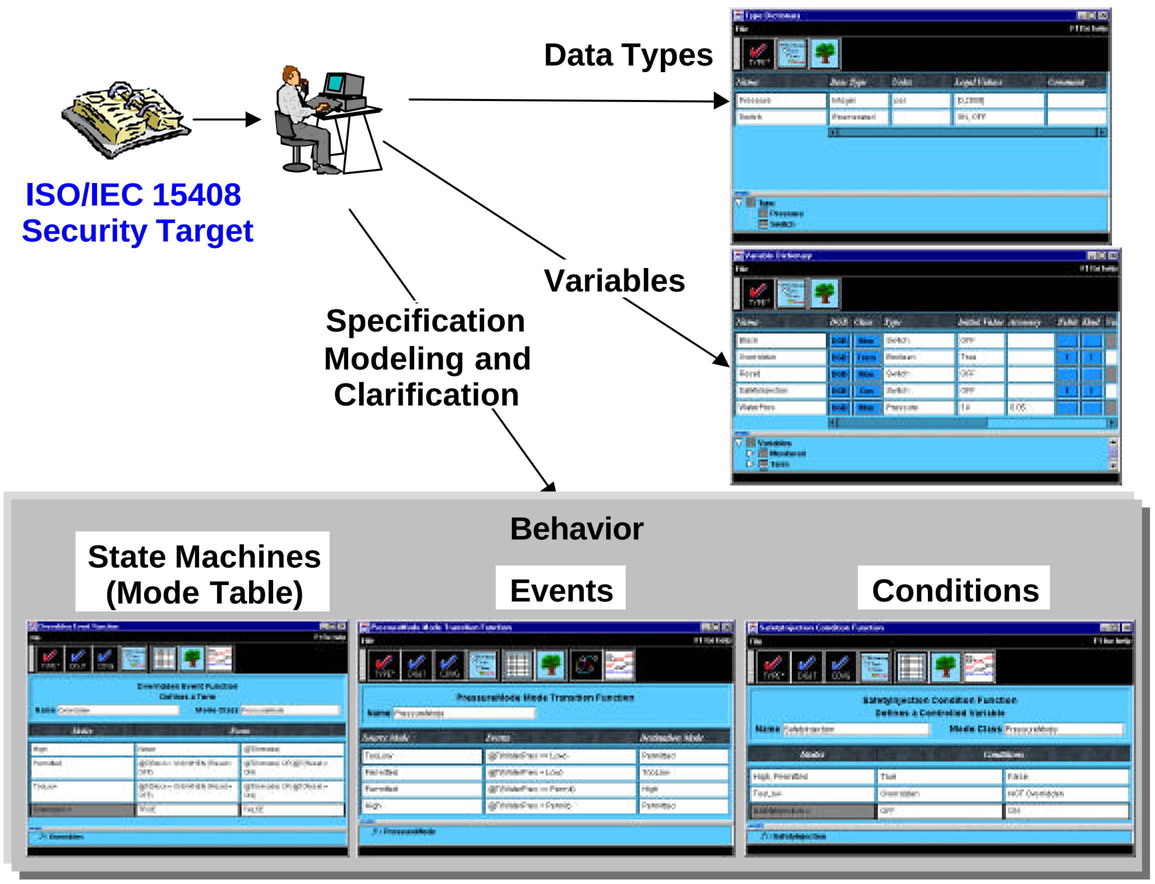


Figure 2. SCR Modeling Constructs

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. This process is typically iterative. It involves defining the variables, the data types associated with the variables, and the tables that define relationships between the variables. A useful guideline for developing SCR models is to work backwards from each output to make the process goal-oriented. The value of each output is defined in terms of the system inputs. Term variables are introduced whenever intermediate values are necessary or useful. The relationships between the inputs and outputs are refined until complete enough to support both manual review and automated analysis. Manual review processes can validate the correctness of the model and completeness with respect to the textual specifications, while automated analysis can identify inconsistencies in the model.

Breaking the GOP specification into clauses supports identifying variables and relationships**.**

Table 1 contains elaboration and clarification of the GOP specifications to support modeling. In addition, it identifies the variables and relationships associated with each clause.

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	valid_grantee
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, grantable	has_grantable_obj_privs

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

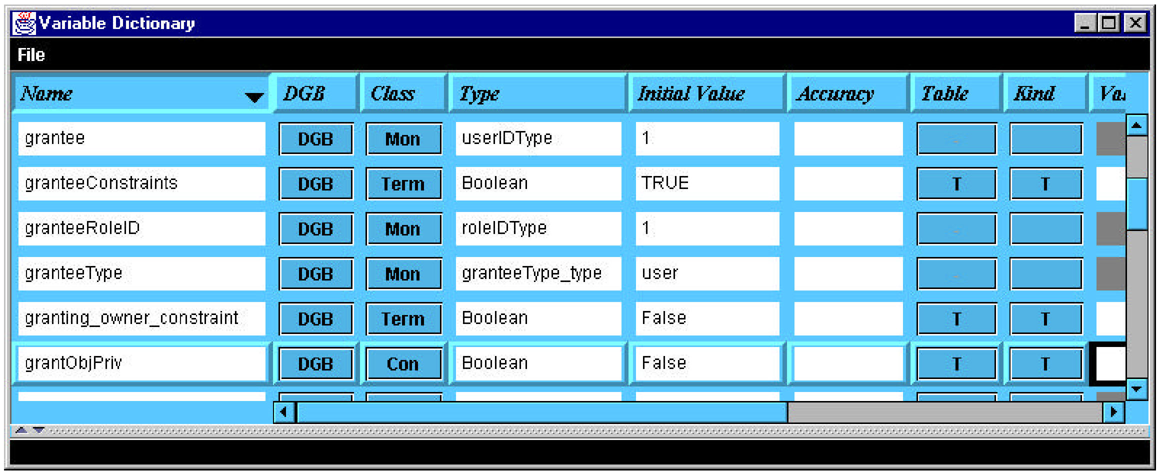
- grantor – user granting an object privilege
- grantee – user being granted an object privilege
- selectedObj – object selected for a particular grant operation
- selectedObjPriv – type of privilege (object access mode) (ALL, SELECT, INSERT, UPDATE, DELETE, etc) on the object that is selected for a 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
- grantable – the flag which indicates whether the privilege (denoted by grantedObjPriv above) the grantor holds on the object (grantedObj above) is grantable to others (users, role or PUBLIC).

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 will be successful. This output is modeled as the Boolean controlled variable:

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

4.3.1 Modeling Variables and Data Types

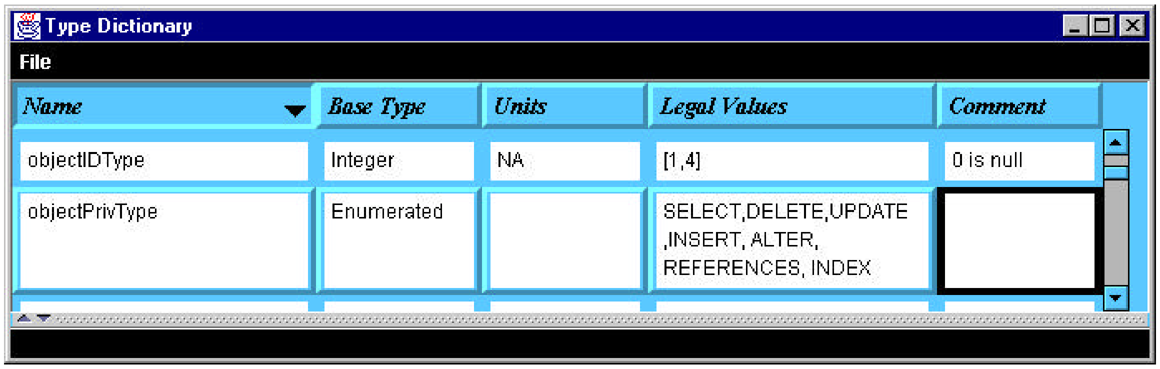
Variables are modeled in the SCRtool through the Variable Dictionary as shown in Figure 3. For example, the grantee is a monitored (input) variable (MON) of type userIDType.



Name	DGB	Class	Type	Initial Value	Accuracy	Table	Kind	Va
grantee	DGB	Mon	userIDType	1				
granteeConstraints	DGB	Term	Boolean	TRUE		T	T	
granteeRoleID	DGB	Mon	roleIDType	1				
granteeType	DGB	Mon	granteeType_type	user				
granting_owner_constraint	DGB	Term	Boolean	False		T	T	
grantObjPriv	DGB	Con	Boolean	False		T	T	

Figure 3. Variables Modeled in SCR

User-defined types are modeled through the Type Dictionary. Data types can be numeric (Integer and Float), Boolean or Enumerated. Figure 4 shows some of the data types used in the GOP model. The type objectPrivType is an enumerated type whose values define valid privileges associated with an object. The type objectIDType is defined as an Integer with a range of 0 to 5. The SCRtool also has a Constant Dictionary for defining constants.



Name	Base Type	Units	Legal Values	Comment
objectIDType	Integer	NA	[1,4]	0 is null
objectPrivType	Enumerated		SELECT,DELETE,UPDATE ,INSERT, ALTER, REFERENCES, INDEX	

Figure 4. Data Types Modeled in SCR

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, terms (intermediate) variables, and mode (state) machines. Figure 5 provides a representation for GOP specification (stated in section 4.2). The output value, grantObjPrivOK, is defined by a condition table referencing three other terms. The specification component GOP(a) is directly associated with the term grantor_owns_object and specification component GOP(b) is directly associated with the term has_grantable_obj_privs. The term valid_grantee is derived from the first sentence in GOP that defines a grantee as a user, role or PUBLIC.

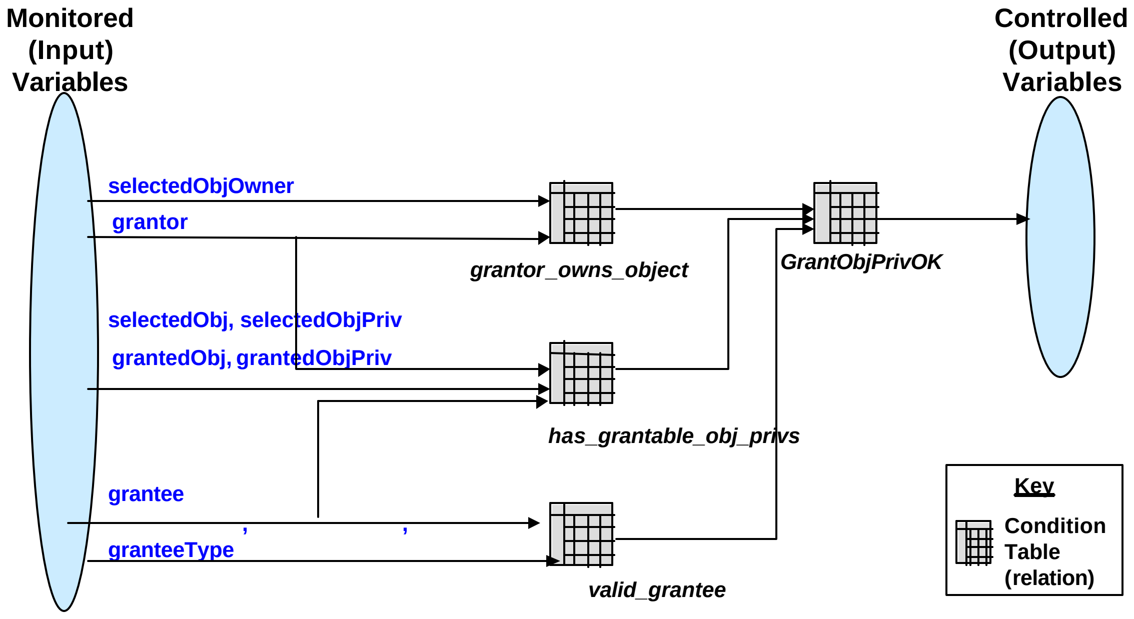

Monitored (Input) Variables
Controlled (Output) Variables
selectedObjOwner
grantor
grantor_owns_object
GrantObjPrivOK
selectedObj, selectedObjPriv
grantedObj, grantedObjPriv
has_grantable_obj_privs
grantee
granteeType
valid_grantee
Key
Condition Table (relation)

Figure 5. Model Structure for Grant Object Privilege

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 grantObjPrivOK.

4.4.1 Modeling the term "grantor_owns_object"

The term grantor_owns_object defines the conditions under which the grantor owns the object for which the he/she is granting a privilege associated with the object to another user (grantee). When these conditions are satisfied, the value of grantor_owns_object is TRUE. The condition table for grantor_owns_object is shown in Table 2. It specifies that the term is TRUE (grantor_owns_object) when grantor = selectedobjOwner, otherwise, the term is FALSE.

Table 2. Condition Table for Term grantor_owns_object

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

The conditions within a condition table can include:

- input or term variables
- arithmetic operators (+,-,*,etc.)
- relational operators (=, !=, >, <, etc.)
- logical operators (AND, OR, or NOT)

4.4.2 Modeling the Term has_grantable_obj_privs

The GOP(b) specification states that if a user wishes to grant an object privilege on an object to another user 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 shown in Table 3 defines these conditions. The term is TRUE when:

1. the selected object is the object for which the grantor holds the privilege (i.e., the selectedObj is the grantedObj).
2. The privilege on the selected object is held by the grantor.
3. The grantor holds the above privilege with the ability to grant to others (GRANT_OPTION is TRUE)
4. the owner of the object is not the grantor
5. the owner of the object is not the grantee

Table 3. Condition Table for Term has_grantable_obj_privs

Table Name	Condition	
	(SelectedObj = grantedObj) AND (SelectedObjPriv = grantedObjPriv) AND (GRANT_OPTION) AND (selectedObjOwner != grantor) AND (selectedObjOwner != grantee)	(SelectedObj != grantedObj) OR (SelectedObjPriv != grantedObjPriv) OR (NOT (GRANT_OPTION)) OR (selectedObjOwner = grantor) OR (selectedObjOwner = grantee)
has_grantable_obj_privs	TRUE	FALSE

The FALSE condition is obtained by negating each of the criteria (1 through 5) given above and joining them together using the connective ‘OR’ thereby implying that the non-satisfaction of even of one of those criteria would make the term ‘has_grantable_obj_privs’ false.

4.4.3 Modeling the Term valid_grantee

The first clause of the GOP specification (See Table 1) specifies that a user, role, or PUBLIC can be granted privileges. Since PUBLIC stands for all the users in the database system, we can identify three classes of grantees – user, role and PUBLIC. These classes are denoted by the variable granteeType.

- If the granteeType is a user, then the grantee should denote a valid user.
- If the granteeType is a role, then the grantee should denote a valid role.
- The granteeType is PUBLIC.

Table 4. Condition Table for Term valid_grantee

Table Name	Condition	
	(granteeType = user AND grantee_is_a_user) OR (granteeType = role AND grantee_is_a_role) OR (granteeType = PUBLIC)	Negation of the Conditions on the left
Valid_grantee =	TRUE	FALSE

The valid_grantee constraint defines condition on variables that must be TRUE for any grant operation to succeed; therefore, conditions for valid_grantee are defined when the output is TRUE.

4.4.3 Modeling the Output grantObjPrivOK

The definition of grantObjPrivOK, shown in Table 3, completes the model for the GOP specification. Its definition includes references to the terms previously described (Table 2, Table3 & Table 4), as well as additional constraints on monitored variables. The two potential values for grantObjPrivOK include:

grantObjPriv = TRUE – test case conditions are such that the privilege can be granted

grantObjPriv = FALSE - test case conditions are such that the privilege cannot be granted

Table 3. Condition Table for Grant Object Privilege (grantObjPrivOK)

Table Name	**Condition**	
GOP(a)	((grantor_owns_object) OR	NOT (grantor_owns_object) AND
GOP(b)	(has_grantable_obj_privs)) AND	NOT(has_grantable_obj_privs) OR
Test Constraints	(valid_grantee) AND (selectedObjPriv_is_valid) AND (grantor != grantee) AND (granteeType = user OR granteeType = role OR granteeType = PUBLIC) AND (((selectedObjType = TABLE OR selectedObjType = VIEW) AND (selectedObjPriv = ALL OR selectedObjPriv = INSERT OR selectedObjPriv = UPDATE OR selectedObjPriv = DELETE)) OR (selectedObjType = PROC OR selectedObjType = FUNC) AND (selectedObjPriv = EXECUTE))	NOT(valid_grantee)OR NOT(selectedObjPriv_is_valid) OR NOT(grantor != grantee) OR NOT(granteeType = user OR granteeType = role OR granteeType = PUBLIC) OR (((selectedObjType = TABLE OR selectedObjType = VIEW) AND NOT(selectedObjPriv = ALL OR selectedObjPriv = INSERT OR selectedObjPriv = UPDATE OR selectedObjPriv = DELETE)) OR (selectedObjType = PROC OR selectedObjType = FUNC) AND NOT(selectedObjPriv = EXECUTE))
GrantObjPrivOK	TRUE	FALSE

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 conditions that ensure that the GOP(a) and GOP(b) conditions are fully exercised during test generation. The conditions ensure the following situations are tested:

- grantee is valid
- the selected privilege is a valid one for the type of object selected
- grantor is not the grantee
- all possible combinations of the granteeType (user, role, or PUBLIC)
- all possible privileges on different types of objects (ALL, UPDATE, SELECT, etc.)

The differences between the TRUE and FALSE case for grantObjPrivOK is that the TRUE case establishes the required conditions:

1. the grantor_owns_object relationship that is associated with GOP(a), where the grantor owns the object, or
2. has_grantable_obj_privs – that is associated with GOP(b)
3. the Test Constraints force complete test coverage of the different grantee types, object types and corresponding privilege types.

The FALSE case establishes the conditions under which the grant operation fails:

1. grantor is not the object owner (i.e., NOT(grantor_owns_object)) AND
2. grantor has not been granted object privilege with the GRANT OPTION (i.e., NOT(has_grantable_obj_privs))
3. At least one test constraint is false.

5 Model Analysis and Test Vector Generation

Modeling and test vector generation is typically performed iteratively as the model is developed. The SCRtool provides a number of checks on the model to ensure that individual tables are consistent and complete. The SCR-to-T-VEC model translator and T-VEC tools perform additional checks that identify cross-table inconsistencies and contradictions. These model analysis capabilities support refining the model by identifying and correcting model defects.

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 as the basis of test vector generation [BBF97]. The translated and compiled version of the grantObjPrivOK requirement includes 20 test specifications. The test vector generator attempts to determine two test vectors for each test specification based on a test selection strategy derived from the concept of **domain testing theory**[2]. Table 4 shows a tabular representation of the 40 test

[2] White and Cohen [WC80] 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. This approach produces test input values that satisfy the conditions of the test specification and that localize the decisions in the specification to maximize defect detection. Once a set of test inputs are selected that satisfy the specification constraints, these inputs are used to derive the value of the output.

vectors produced for grantObjPrivOK. The test vectors include 12 monitored variables and 6 term variables (not shown in the table). The test values shown in Table 4 reflect how the test generator systematically selects low-bound and high-bound test points at the domain boundaries. The input values 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 privName and granteeType.

Table 4. Test Vectors for grantObjPrivOK

Vector #	DCP	grantOObjPriv	grantor	grantee	privName	grantee Type	objOwner	selected Obj	granted Object	GRANT_ OPTION	grantee RoleID	roleID
1	1	TRUE	1	2	ALL	user	1	4	4	TRUE	2	2
2	1	TRUE	4	3	ALL	user	4	1	1	FALSE	0	0
3	2	TRUE	1	2	UPDATE	user	1	4	4	TRUE	2	2
4	2	TRUE	4	3	UPDATE	user	4	1	1	FALSE	0	0
5	3	TRUE	1	2	SELECT	user	1	4	4	TRUE	2	2
6	3	TRUE	4	3	SELECT	user	4	1	1	FALSE	0	0
7	4	TRUE	1	2	INSERT	user	1	4	4	TRUE	2	2
8	4	TRUE	4	3	INSERT	user	4	1	1	FALSE	0	0
9	5	TRUE	1	2	DELETE	user	1	4	4	TRUE	2	2
10	5	TRUE	4	3	DELETE	user	4	1	1	FALSE	0	0
▪ ▪ ▪												
37	19	FALSE	1	2	DELETE	user	3	1	1	FALSE	0	1
38	19	FALSE	4	3	DELETE	user	2	4	4	FALSE	2	1
39	20	FALSE	1	2	DELETE	role	3	1	1	FALSE	1	1
40	20	FALSE	4	3	DELETE	role	2	4	4	FALSE	2	2

The number of vectors generated and the specific test values depend on the test vector generation mode, test input selection heuristics, and the satisfiability of the test specification conditions. A test specification is considered satisfiable, if a set of input values exist that satisfy all conditions and result in a valid expected output value. Unsatisfiable test specifications typically result from specification errors .

6 Test Driver Generation and Execution

The last step in the process involves transforming the tests into a test driver that can be executed against a product, which in our case is the Oracle DBMS product Version 8.05. NIST stated that the capability to transform models into test drivers for a variety of platforms is an important discriminating capability of this toolset.

The test driver generator combines test driver schemas, user-defined object mappings and test vectors to produce test drivers as illustrated in Figure 6. The test driver schema encodes generic descriptions for test execution based on an algorithmic pattern that is applicable to the specific test environment. The object mappings relate objects in the model to the objects in the implementation or component interfaces. 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
- Set the values of the test inputs

- Cause execution of the test
- Retrieve and save the results of the test execution

Test driver schemas provide a description of how to accomplish each of these steps for a specific testing environment using a small language that can access information about the specification model, data objects, types, ranges, test values, and user customizable information. A schema is also used to describe the form of expected outputs to support test execution and results analysis.

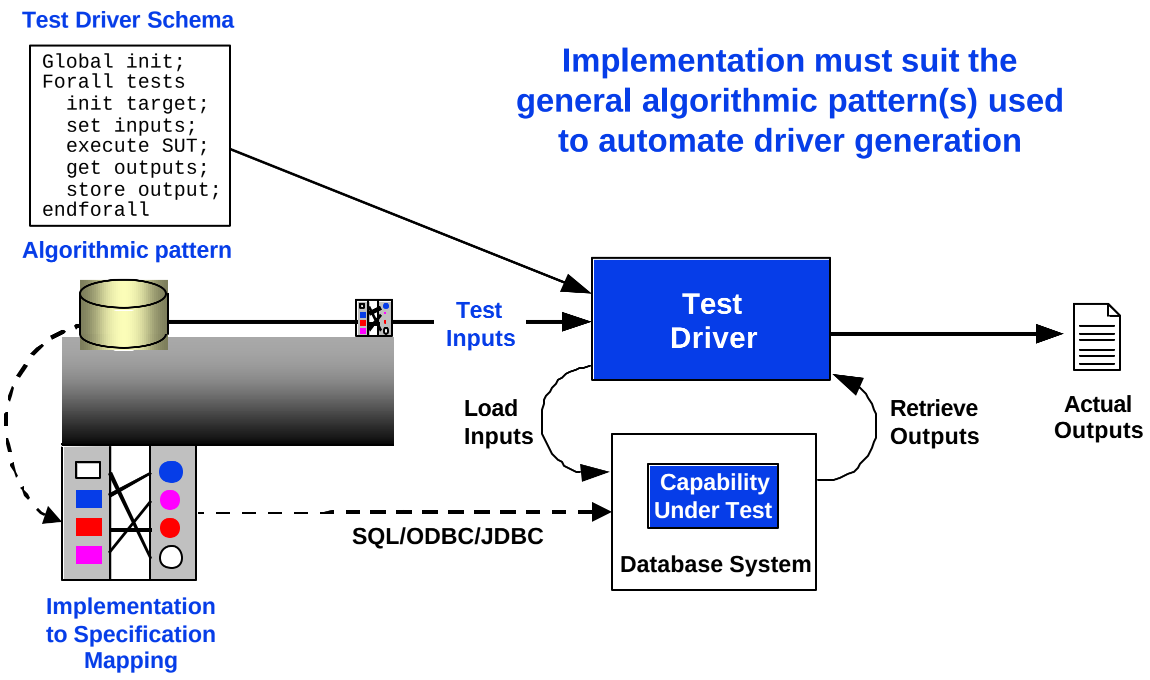


Figure 6. Elements of a Test Driver

Two different test driver schemas and object mapping descriptions were used with the grantObjPrivOK model to test two different test environments. The first test environment involved generating test drivers for testing the InterBase 6.0 DBMS, and the second environment involved testing the Oracle 8.05 database engine. 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.

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 were modeled using the SCRtool. These models were then used as the basis of automated test vector and test driver generation with the T-VEC toolset for multiple product applications and test environments. This approach reduces the time and effort associated with security testing, while increasing the level of test coverage. NIST cited the approach's ability to support driver generation for a variety of platforms as a key discriminator. These results demonstrate the feasibility of using model-based test automation to improve the economics of security functional testing. Specifically, the TAF approach is applicable to security evaluation laboratories and other commercial organizations that need a cost-effective approach for performing security functional testing.

7.1 Other Applications and Results

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 [BB96]. Statezni described how the approach supports requirement-based test coverage mandated by the FAA with significant life cycle cost savings [Sta99; Sta2000]. Safford presented results stating the approach reduced cost, effort, and cycle-time by eliminating requirement defects and automating testing [Saf2000]. 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 some hardware/software integration functionality. It has been applied to critical applications like telemetry communication for heart monitors, flight navigation, guidance, autopilot logic, display systems, flight management and control laws, airborne traffic and collision avoidance. In addition, it has been applied to non-critical applications such as workstation-based Java applications with GUI user interfaces and database applications. 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, COTS test injection products, and test environments.

7.2 Future Work

The development team continues to evolve the model translation capabilities to support functional, object-oriented, control system and hybrid modeling approaches. In addition, the team is involved in the Object Management Group, UML Action Language Semantics formalization. The team is also involved in the development of modeling guidelines and training material that help integrate commercial modeling approaches with verification tools.

As continued support for NIST, additional models for the Oracle Security Target are being modeled to address the testing of security services relating to audit generation, security management, identification, authentication, and session management.

8 References

[BB96] Blackburn, M.R., R.D. Busser, T-VEC: A Tool for Developing Critical System. In Proceeding of the Eleventh International Conference on Computer Assurance, Gaithersburg, Maryland, pages 237-249, June, 1996.

[BBF97] 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, Gaithersburg, Maryland, pages 54-67, June, 1997.

[Bla98] Blackburn, M. R., Using Models For Test Generation And Analysis, Digital Avionics System Conference, October, 1998.

[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.

[Ora00] Oracle Corporation, Oracle8 Security Target Release 8.0.5, April, 2000.

[Sta99] Statezni, David, Industrial Application of Model-Based Testing, 16th International Conference and Exposition on Testing Computer Software, June 14-18, 1999.

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

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

[WC80] White, L.J., E.I. Cohen, A Domain Strategy for Computer Program Testing. IEEE Transactions on Software Engineering, 6(3):247-257,May, 1980.