Applying the Test Automation Framework With Use Cases and the Unified Modeling Language

Rich McCabe
Systems and Software Consortium, Inc.

Mark Blackburn
Systems and Software Consortium, Inc.

Abstract

Currently, there is great interest in the use case technique and the Unified Modeling Language (UML). This paper explains how use cases and UML models are somewhat limited in the information they provide to testers. Any effective set of tests represents a detailed understanding of the requirements of the unit under test, as well as the interfaces provided to the test drivers. Although use cases and UML diagrams provide many important clues to the tester, they typically lack much vital information, especially in support of rigorous and cost-effective testing, as accomplished with the Software Productivity Consortium's Test Automation Framework (TAF). Given these limitations, the focus is on how best to employ TAF when working with UML artifacts.

This is the first Consortium paper addressing the issue of using UML to support test automation with TAF. Based on member requests, this paper may be extended to include more technical details explaining how typical UML artifacts may be used to provide interface information.

Contents

Introduction

Why This Paper Is Worth Reading

This paper explains how to employ use cases and Unified Modeling Language (UML) artifacts with the Software Productivity Consortium's Test Automation Framework (TAF) in system/software development. In particular, the discussion covers use cases and use case

diagrams, interaction diagrams, class diagrams, and statechart diagrams, as well as describing how they are related to system testing, integration testing, and unit testing.

Reliable, safe, secure, and correct software is of greater concern than in the past as systems grow ever more complex. The strategy for system verification and validation is a crucial aspect in addressing these concerns. The cost of testing is increasing as the size and complexity of systems increase. TAF has been demonstrated to support rigorous requirements analysis and automated testing in a timely, cost-effective manner. TAF reduces defects, risk, and testing schedule and cost—up to 90 % of schedule (Software Productivity Consortium 2000) and 40 to 60 % of test/verification costs (Kelly et al. 2001; Safford 2000).

UML is increasingly used as the notation of choice in software development and is featured in many tools and methods, such as the Consortium's Object-Oriented Approach to Software-Intensive Systems (OOASIS) method. Despite widespread use of UML, there are frequent misunderstandings in how to employ it for testing, especially automated testing as supported by TAF. Binder (2000), in his comprehensive summary of the application of UML to testing, points out the limitations of UML in testing: "UML does not provide explicit support for combinational logic and domain definition, which are essential for test design. As a practical matter, the models (whether using UML or not) produced for many systems need much work before test cases can be developed from them." This paper deals with the pragmatics of applying TAF using UML artifacts given this typical situation.

Scope

The focus of this paper is on system and integration testing. As Binder (2000) notes, there are at least three potential levels of testing in a software-intensive system:

- System test involves a complete integrated application, where the tests focus on capabilities or characteristics that are present only with the entire system.
- Integration test involves a complete system or subsystem where tests exercise interfaces among units within the specified scope to demonstrate that the units are operable collectively. Test strategies for large systems may involve independent testing of its subsystems or other collections of system components short of the whole system.
- Unit test typically is performed by the developer on units of functionality too small to be individually specified by system designers and verified by independent test groups.

Although TAF has been used for unit testing, it is not addressed here.

This paper provides a brief overview of TAF and discusses how various artifacts may be mapped into TAF verification models to support automated analysis and test generation. In addition, TAF model development from use cases is illustrated with an example.

The reader is presumed to have some familiarity with the following:

- UML and, in particular, use case diagrams, interaction diagrams, and statechart diagrams (the current standard is UML 1.4)
- Object-oriented (OO) software and related concepts (although objects and classes are mentioned only in passing)

See the section For More Information for resource suggestions on these topics.

TAF

TAF is a method and associated toolset to support model development, model analysis, and model-based test automation. TAF supports modeling methods that focus on representing requirements, like the Software Cost Reduction (SCR) method, as well as approaches that focus on modeling both requirements and design (as in The MathWorks' Simulink®). TAF integrates various government and commercially available model development and test generation tools to support defect prevention and automated testing of systems and software. Unfortunately, as of this writing, UML is not defined with sufficient semantics to enable full analysis and automated test generation (Fontaine and Blackburn 1998).

Currently the T-VEC® Test Vector Generation System is the TAF toolset used for test generation. In addition, T-VEC supports test driver generation, requirement test coverage analysis, and test result checking and reporting. Through model translation, requirement-based or design-based models are converted into T-VEC test specifications, from which T-VEC derives test vectors. These test vectors include test inputs and expected test outputs, as well as model-to-test traceability information. T-VEC's test driver generation capability supports preparing the test vectors for execution in the test environment.

The most typical application of TAF is a method based on interface-driven requirements modeling, which is applicable to most systems, including those developed with UML and OO. With this approach, test engineers work in parallel with developers to stabilize interfaces, refine requirements, and build models to support iterative testing and development. The following outlines the process, as depicted in Figure 1:

1. Working from whatever requirements artifacts are available, testers create a rigorous verification model[1] in a table-based format[2] using a tool based on the SCR method, such as the SCRtool or T-VEC Tabular Modeler (TTM). Simplistically, each table represents an output, specifying the relationship between input values and resulting output values. T-VEC automatically checks for inconsistencies in the model. The tester interacts with the requirements engineers to validate the model as a complete and correct interpretation of the requirements.

2. The tester maps the variables (inputs and outputs) of the verification model to the interfaces of the system. The nature of these interfaces depends on the level of testing performed. At the system level, the interfaces may include graphical user interface widgets, database application programming interfaces (APIs), or hardware interfaces. At the lowest level they can include class interfaces or library APIs. The tester uses these mappings to build test driver templates to support automated test driver generation. The tester works with the designers to ensure the validity of the mappings from model to implementation.

[1] "… a **verification model** specifies behavioral requirements in terms of the interfaces for the system under test. This is in contrast to a "pure" requirements model, which specifies the requirements in terms of *logical entities* representing the environment of the system under test. 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." (Blackburn et al. 2000)

[2] Although analysts and designers commonly develop models based on state machines or other notations, testers apparently find it easier to learn and develop requirements for test in the form of tables. See Blackburn et al. (2001) and Kelly et al. (2001) for more information.

3. The T-VEC tool generates an optimal set of test vectors by analyzing the input space. Based on the constraints defined in the model, it identifies boundary conditions most likely to expose defects in the system response. T-VEC generates the corresponding test drivers (using the test driver templates) and executes the test drivers in the target or host environment. The test drivers typically are designed as an automated test script that sets up the test inputs enumerated in each test vector, invokes the unit under test, and captures the results.

4. Finally, T-VEC analyzes the test results. It compares the actual test results to the expected results and highlights any discrepancies in a summary report.

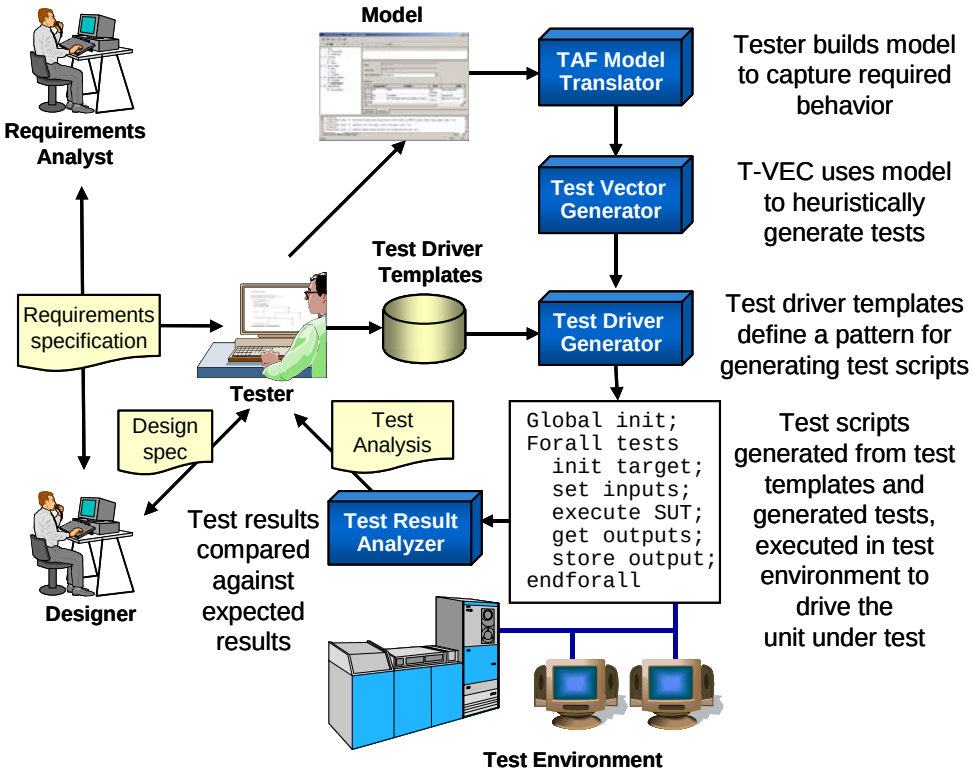

Figure 1. Typical TAF Process

What TAF Requires

The key prerequisite for test design, whether done manually or through automated test selection and generation, is defining the relationships between input and output values. As with any approach to test design, TAF requires two primary inputs:

- Functional requirements, which describe the required behavior of the unit, component, or subsystem under test (a requirement specification)
- Interface definitions, which describe the inputs and outputs, their data types and range information, and the facilities for interacting with the unit under test

The system or software design must permit testers to create valid test driver templates, from which T-VEC generates working test drivers to run test scripts in the test environment. This means that interfaces must be available for injecting test inputs (including setting internal state

as needed), executing the system under test, and observing the test results. Test scripts act on behalf of multiple test cases to:

- Initialize the unit under test
- Set inputs and execute the unit
- Retrieve and store the results

Usually some but rarely all information needed to apply TAF is available in use case documentation and UML artifacts. The next section describes the problems in more detail and provides some guidance.

Building a TAF Verification Model

The point of testing is to verify that the system or component under test meets its specification (i.e., requirements). Depending on project conventions and the system under test, various artifacts are used to represent the specifications. Many projects use various UML artifacts with other types of models and often precede OO design at the component or subsystem level with functional analysis and architectural design at the system level. Regardless, this paper deals with the following artifacts, consistent with its focus on UML:

- Use cases and use case diagrams
- UML statechart diagrams
- UML interaction diagrams
- An abstract class or interface specification (in code or attached to a UML interaction or class diagram)

Even these artifacts are interpreted and employed variously by different projects. Recognize also that a given tool, although UML-based, may not support all legal constructs in UML (but may, on the other hand, offer its own unique extensions to UML, none of which will be addressed here, of course).

Use Cases and Use Case Diagrams

Use cases are often used to document the highest level of system behavior; however, larger projects tend to treat subsystems as systems in their own right and consequently may apply the use case technique recursively to subsystems, subsubsystems, and the like.

Use case diagrams are a UML standard but are no more than a depiction of the gross structure of the behaviors demanded of the system. Although they may contain a few clues for the verification model, they are hardly more than a “table of contents” for the use cases.

The UML standard does not cover use cases. As commonly practiced, use cases are formatted as structured text. Projects often adopt a common template for use cases. Some of the most typical sections of a use case include the following:

- Name/goal
- Actors
- Preconditions (and postconditions)
- Main scenario
- Alternate courses

The actors designate the input sources and output sinks for the unit under test. This information may help to clarify the nature of an input or output that should be included in the verification

model. In general, tests should cover system interactions with all actors; thus, the tests will involve all inputs and outputs.

The preconditions are a strong clue to a system mode or state, sometimes expressed in terms of one or more use cases that must occur (or execute successfully) before the use case at hand may even be invoked (e.g., the Recognize Card use case in the example below). In terms of the verification model, this indicates an input condition that must be set as part of the setup for tests related to this use case. Sometimes a use case also specifies postconditions, indicating under what circumstances the execution of the use case results in a change in system state (i.e., a test output). System states therefore appear as variables in the verification model and are used as both inputs and outputs. Generally each state is a separate variable (with values “active” and “not active” or the equivalent), except for states that are logically exclusive (e.g., the system can only be in one of the following states at any given time: take off, landing, reconnaissance, or combat).

The main scenario is the primary “trace” or stimulus/response behavior represented by the use case, often presented as the simplest or most natural scenario in which the system accomplishes the goal or essential behavior of the use case. The alternate courses are subsidiary paths that branch off from the main scenario, dealing with more complex success scenarios or failure cases and handling “unexpected” events. The scenarios are written as a partially ordered sequence of steps. While not practiced universally, the best technique is to treat the system as a black box, so that each step is either an input to the system or an external output from the system. Without black box treatment, use cases begin to resemble UML interaction diagrams, in that they distinguish the interactions of internal components, but the tester can still use them as described in the next section by simply ignoring the internal interactions (i.e., ignoring interactions internal to the unit under test).

The tester, by walking through the use cases, can infer variables of the verification model and begin to build the model incrementally. Each step represents some input or output, but the informality of use cases leads to the following difficulties:

- Distinguishing the same variable in multiple steps and across multiple use cases. The same variable is likely to appear in multiple places but is not necessarily named consistently. In fact, it may not be named at all, but is only implicit in some action. This is where reference to the interface specification proves invaluable; however, interaction with both analyst and designer is also crucial.
- Relating specific ranges of input and output values. Use cases typically do not include extensive annotations detailing logical and algorithmic relationships among inputs and outputs. By context, the tester usually may infer that a relationship exists but not the specific contents of the data. Of course, for some early testing, the detailed values may not be important (e.g., testing priority is to check that the system successfully transmits a report to the right destination, not the contents of the report). Rigor usually demands detail, however, and again the tester will need to refer to the designer as well as the design specifications.
- Distinguishing state changes within a use case. There is no guarantee that significant system state changes impacting the expected output will be matched one to one with use cases. Multiple state changes may occur over the course of a single use case. Such states may be inferred from the use case, but most likely they will be obvious only to the analyst or designer. Without a statechart diagram or equivalent information in the interface specification, the tester is most likely to discover such states during discussions with analysts or designers to uncover the value relationships among inputs and external outputs.

Without interface specifications to reference, the testers should collaborate with the designers to identify model variables that map well to the existing system interfaces and/or to establish test-friendly interfaces. While testers can manage these mappings (from verification model variables to interface elements) at a single point of change and define them to be arbitrarily complex, the mappings do not wholly mitigate the risk of a poor model-to-interface match. The tester achieves best results by working with the designer to ensure that the interface will support a straightforward mapping between model and interface, as well as facilities to support the other needs of the test drivers.

Interaction Diagrams

Interaction diagrams are either sequence or collaboration diagrams, but they are semantically equivalent for the purposes of verification modeling. The interaction diagram depicts objects, components, and/or subsystems of the system interacting, often to realize a specific use case (or a particular scenario or trace of a use case).

The tester then would consult interaction diagrams when the unit under test is a component of a larger system, focusing on this component's interactions with its siblings. The same component appears in multiple diagrams, and the tester must peruse all of them in order to identify the same inputs and outputs within different scenarios. Again, as in the other diagrams, the specification of those inputs and outputs may be detailed or left vague. Regardless, the tester requires additional information from the designer and relevant interface specifications, usually not only for the unit under test but also for the other components interacting with the unit under test. Interaction diagrams typically do not provide any information relating input to output value ranges, and neither are system states represented explicitly.

As with use cases, the tester should walk through each interaction diagram. Here the tester focuses on the unit under test but should absorb the whole diagram to get the context of the interaction and infer any relevant system state. If the interaction diagram does correspond to a use case, then the use case can provide clues to pertinent system states, as described in the earlier section on use cases.

Statechart Diagrams

Statechart diagrams, the UML notation for finite-state machine models, are the UML diagrams closest to meeting the needs of a verification model, but still lack crucial information. Efforts are ongoing to augment UML statechart diagrams with an "action language,"[3] but this has yet to be standardized.

Statechart diagrams are somewhat more convenient than use cases in that all the significant system states are explicitly represented, as well as their relationships to inputs and outputs. Furthermore, the conciseness of the notation helps the tester peruse the diagram, collecting inputs and outputs; however, complicated systems still have large, complicated, multilevel statechart diagrams.

The tester still must recognize and input and output variables as they appear in multiple places throughout the diagram and map these to actual constructs in the code. In fact, the very terseness of the diagram may make it more difficult for the tester to infer any missing information. Statechart diagrams often are inconsistent or silent in identifying inputs and outputs

[3] An action language is a formal notation for representing the effects of actions in the context of some model (for example, a finite-state machine).

and their values; an input (or output) may be designated by its value, name/type, or different things in different places. The analyst's intent may be more obscure than in a goal-directed use case.

Interface Specifications

As has been reiterated throughout this paper, interface specifications are crucial to successful application of TAF, as well as any test design. All software components eventually have an interface specification, even if it is only the interfaces as defined in the code itself.

UML provides for interface specifications of classes, components, and subsystems. The UML interface specification is a set of operation signatures, describing the same essentials as any programming language. Operation signatures minimally provide an actual set of inputs and outputs, specifying data types for parameters and return values.

While verification model variables must map to these actual inputs and outputs, the mapping is not necessarily one-to-one. Still, the tester should examine available interfaces closely to confirm that abstractions seeming to appear in the requirements do exist in the code and that model variables and state values can in fact be mapped to these interfaces. The interface specification does not provide all the information the tester needs to produce the verification model, but it can help guide the tester to ask the right questions from the designer and analyst.

ATM Example

The following simplified example of an automated teller machine (ATM) is used to illustrate primarily how UML use cases can be mapped to TAF models. The example concludes with some discussion of the other artifacts.

Figure 2 depicts a use case diagram for the ATM. Admittedly, it shows only a partial and simplified set of use cases compared to those for a real ATM. Note that the example treats the bank accounting system as part of the ATM system rather than as an external actor.

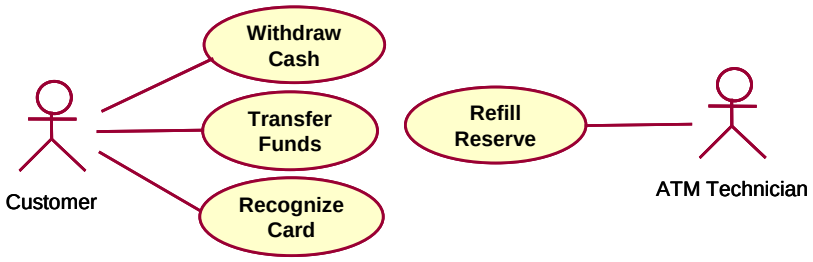

Figure 2. Use Case Diagram—ATM Example

To support testing, as discussed earlier, the required behavior must be identified and expressed in terms of some interface. From a high-level perspective, the use cases denote various broad behaviors but this is not sufficient for building test cases or models. More detailed information is required to formalize the interfaces and required behavior.

Use Case

Table 1 specifies a portion of the use case for withdrawal. The portion shown omits a great many alternate courses that would be present in the complete use case, such as handling timeouts, the consequences of an invalid personal identification number (PIN), and foreign language options.

Table 1. Withdraw Cash Use Case (Extract)

Name/Goal	Withdraw Cash
Actors	Customer
Preconditions	1. The Recognize Card use case has completed successfully.
Postconditions	1. Current balance = original balance – dispensed cash – service charge. 2. Cash reserve = original reserve – dispensed cash.
Main scenario	1. Customer requests cash withdrawal. 2. ATM asks Customer for amount. 3. Customer inputs desired amount. 4. ATM dispenses amount of cash requested, card, and receipt.
Alternate courses	2a. Customer card linked to multiple accounts. 2a1. ATM asks Customer to select one account (savings, checking, or credit) for the transaction. 2a2. Customer selects an account. 2a3. Go to Step 2. 2b. Customer card linked to nonaffiliate account. 2b1. ATM asks whether Customer will accept $2 service charge. 2b2. Customer accepts charge. 2b3. Go to Step 2. 2c. Card linked to credit account. 2c1. ATM asks whether Customer will accept $10 service charge. 2b2. Customer accepts charge. 2b3. Go to Step 2. 4a. Customer requested amount not a multiple of $20. 4a1. ATM asks Customer to enter a multiple of $20. 4a2. Go to Step 3. 4b. ATM has less cash in reserve than requested amount. (etc.)

This use case provides enough information to begin creating a verification model, albeit at some risk that the variables of model will not map to system interfaces as ultimately coded (see discussion at the end of the section Use Cases and Use Case Diagrams).

Verification Model

Analyzing the Withdraw Cash use case (and the other use cases not shown in detail in the table) suggests a number of abstract input variables to use in the verification model:

- Amount: The amount specified for a withdrawal or transfer
- Balance: The balance of the account prior to the transaction

- Cash_reserve: The amount of cash in the ATM machine prior to the transaction
- Institution: The financial institution type: AFFILIATE or NON_AFFILIATE
- Transaction: The type of transaction: WITHDRAWAL, TRANSFER, CREDIT

In addition, the associated data types and domains (i.e., possible range of values in that data type) may be determined, as presented in Table 2. Again, these variables are somewhat speculative in that they may not map well to the actual system interfaces as coded.

Table 2. Variables for Withdrawal Use Case

Input Variable Name	Data Type	Domain
Amount	amount_type	0,999999
Balance	amount_type	0,999999
Cash_reserve	cash_reserve_type	0,10000
Institution	institution_type	AFFILIATE,NON_AFFILIATE
Transaction	transaction_type	WITHDRAWAL,TRANSFER,CREDIT

A sample model of one of the withdrawal requirements is represented in Table 3 as a condition table. A condition table specifies values and constraints for the outputs. The condition table shown in Table 3 defines the conditions that limit the cash dispensed by the ATM. These conditions rigorously express those requirements described more informally in the Withdraw Cash use case.

Table 3. Requirement Model for Cash Dispensed in the Withdrawal Use Case

Assignment	Condition
(Amount)	(Transaction = WITHDRAWAL OR Transaction = CREDIT) AND (Amount = 20 OR Amount = 40 OR Amount = 60 OR Amount = 80 OR Amount = 100 OR Amount = 120 OR Amount = 140 OR Amount = 160 OR Amount = 180 OR Amount = 200 OR Amount = 220 OR Amount = 240 OR Amount = 260 OR Amount = 280 OR Amount = 300) AND (Amount <= Cash_reserve) AND (t_funds_available)
(0)	((Transaction = WITHDRAWAL OR Transaction = CREDIT) AND (NOT(Amount = 20 OR Amount = 40 OR Amount = 60 OR Amount = 80 OR Amount = 100 OR Amount = 120 OR Amount = 140 OR Amount = 160 OR Amount = 180 OR Amount = 200 OR Amount = 220 OR Amount = 240 OR Amount = 260 OR Amount = 280 OR Amount = 300)) OR (Amount > Cash_reserve) OR NOT (t_funds_available))

Note the condition table for withdrawal depends on the term variable t_funds_available, which is shown in Table 4. In the TTM tool, a term variable can be referenced as part of the constraints or value calculations of other inputs, terms, or output variables. They reduce the complexity of the model by simplifying expressions and eliminating redundancies. For example, the condition table for t_funds_available references term t_service_charge. The condition table for t_service_charge is shown in Table 5 and defines the value and conditions of the service.

Table 4. Requirement Model for Term t_funds_available

Assignment	Condition
(TRUE)	(Balance - Amount - t_service_charge) >= 0
(FALSE)	(Balance - Amount - t_service_charge) < 0

Table 5. Requirement Model for Term t_service_charge

Assignment	Condition
2	Institution = NON_AFFILIATE AND (Transaction = WITHDRAWAL OR Transaction = TRANSFER)
(0)	Institution = AFFILIATE AND (Transaction = WITHDRAWAL OR Transaction = TRANSFER)
10	Transaction = CREDIT

The T-VEC Test Vector Generation System calculates an optimal set of test vectors based on all of the condition tables shown earlier. A sample of the generated test vectors is shown in Table 6. From a total of 60 test vectors, only a portion of the tests are shown in the table.

Table 6. Test Vectors for Withdrawal Use Case

#	__output	Amount	Balance	Cash_reserve	Institution	t_funds_ available __VAR	t_service_ charge __VAR	Transaction
1	20	20	22	20	NON_AFFILIATE	TRUE	2	WITHDRAWAL
2	20	20	999999	10000	NON_AFFILIATE	TRUE	2	WITHDRAWAL
3	40	40	42	40	NON_AFFILIATE	TRUE	2	WITHDRAWAL
4	40	40	999999	10000	NON_AFFILIATE	TRUE	2	WITHDRAWAL
[Skip rows]								
58	280	280	999999	10000	AFFILIATE	TRUE	10	CREDIT
59	300	300	310	300	NON_AFFILIATE	TRUE	10	CREDIT
60	300	300	999999	10000	AFFILIATE	TRUE	10	CREDIT

Other Artifacts

An interaction and/or class diagram will begin to provide the tester with the expected interfaces. The interaction diagram may expose other system behavior not detailed in (or not apparent in) the use cases. The tester can improve and expand the initial model shown above based on this information or deal with minor mismatches by correcting the definitions of model-to-interface mappings. The following paragraphs discuss a few of the typical problems a tester would encounter.

The data types of the model variables may be different from those of the corresponding variables in the system implementation. For instance, the Balance and Amount variables actually may be typed as decimal values (i.e., dollars and cents).

Some input variables may not map to an actual variable in the code and must be controlled indirectly. For example, the Institution variable may be best controlled by setting the card identifier (a variable arising from the Recognize Card use case not explicitly called out in the example) linked to an account associated with an institution of the desired type. Test drivers may need to construct preliminary transactions to set the Balance variable correctly during test setup.

The system may implement other model variables as multiple variables or, in the worst case, demand a reconception of some model variables and the consequent reworking of dependent condition tables. As a trivial example, the system may store the state of the cash reserve as an integer representing the number of bills on hand, rather than as a dollar value. Balance may not be directly accessible; for example, it could be embedded in the receipt as text and is best checked via indirect means, such as the result of subsequent transactions.

Any statechart diagrams also may serve as both a check and an expansion of the verification model. They should be concisely and definitively represent system states that could only be inferred beforehand by examining diverse portions of the use cases or interaction diagrams. These logical states, however, may not map simply to values stored in the system implementation. For example, the Card Recognized state, a necessary precondition for the Withdraw Cash use case, may be represented in the system only by having reached a certain location in the program where the Institution variable (as well as other variables such as Card_number and PIN) has been set correctly.

Conclusions

This paper has shown how TAF may be applied successfully by working from use cases and UML models, although the task is certainly not mechanical or trivial. The tester must do the mental work of translating less-than-rigorous UML specifications into a rigorous verification model that maps successfully to the system implementation. The tester requires close interaction with both analysts and designers to supply crucial information missing from the UML artifacts.

Future Work

This paper is not a comprehensive treatment of the issues related to applying TAF in OO development projects. Based on member requests, the Consortium may continue to extend its explorations in this area to address the following:

- A full tutorial for building TAF models from UML artifacts that illustrates how UML information can support the definition of interfaces that are critical to the development of models
- OO design and code techniques to build test driver templates and map verification model variables to code interfaces
- Employing UML version 2.0 (and succeeding versions) with TAF
- TAF use with particular commercial UML-based modeling tools

Furthermore, similar challenges abound in applying TAF using the artifacts typical of a functional approach to analysis and design (for example, behavioral diagrams).

For More Information

Members with general questions or comments on any of the topics in this paper or related topics, or members interested in applying TAF or OO technologies with Consortium assistance, should contact the authors or their member account director (see http://www.software.org/pub/keycontacts.asp).

For more on TAF, see the Consortium's TAF Website at http://www.software.org/pub/taf/testing.html or contact the authors.

For more about the SCR method, see Heitmeyer, Jeffords, and Labaw (1996). In addition, there are examples of the method on the TAF Website at http://www.software.org/pub/taf/Reports.html.

OOASIS is the Consortium's methodology for system/software development; it is an example of a UML-based approach that fits well with TAF using the techniques described in this paper. For more information, visit the OOASIS Website[4] at https://www.software.org/membersonly/ooasis/ or contact the authors. For specific guidance on the use case technique, look to https://www.software.org/membersonly/OOASIS/tasks/capture_behavioral_requirements.asp. Many books cover the use case technique, but Cockburn (2000) is one of the best.

For a quick overview of UML diagrams, see https://www.software.org/membersonly/OOASIS/umlreference.asp[4]. For a more complete introduction to UML, Fowler and Scott (1999) is a good source. The Object Management Group (at http://www.omg.org/) is the industry consortium that owns and updates the UML standard.

For an introduction to OO concepts, consider Page-Jones (2000) as a good choice among the many available texts.

References

Binder 2000	Testing Object-Oriented Systems—Models, Patterns and Tools. Reading, Massachusetts: Addison-Wesley.
Blackburn, M.R., R.D. Busser, and A.M. Nauman 2000	Interface-driven Model-Based Test Automation. In *Proceedings of the Software Testing Analysis and Review Conference* (STARWEST 2000).
Blackburn, M.R., R.D. Busser, and A.M. Nauman 2001	Removing Requirement Defects and Automating Test. STAREAST, May 2001.
Cockburn, A. 2000	*Writing Effective Use Cases.* Reading, Massachusetts: Addison-Wesley.
Fontaine, J., and M. R. Blackburn 1998	*Automatic Test Generation Support for Object Technologies,* SPC-98068-MC, version 01.00.00. Herndon, Virginia: Software Productivity Consortium.
Fowler, M., and K. Scott	*UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language.* Reading, Massachusetts: Addison-Wesley.

[4] This page requires that you have a member account to gain access, but it is free to all members. Request an account at https://www.software.org/catalog/welcome_new_member.asp.

1999	
Heitmeyer, C., R. Jeffords, and B. Labaw 1996	Automated Consistency Checking of Requirements Specifications. *ACM TOSEM* 5(3):231-261, 1996. Also available at http://chacs.nrl.navy.mil/publications/CHACS/1996/1996heitmeyer-ACM.pdf.
Kelly, V., E. L. Stafford, M. Siok, and M. R. Blackburn 2001	Requirements Testability and Test Automation. Lockheed Martin Joint Symposium, June 2001.
Page-Jones, M. 2000	*Fundamentals of Object-Oriented Design in UML.* Reading, Massachusetts: Addison-Wesley.
Safford, E. L. 2000	Test Automation Framework, State-Based and Signal Flow Examples. *Twelfth Annual Software Technology Conference*, 30 April to 5 May 2000.
Software Productivity Consortium 2000	Rockwell *Pilot Project Technical Note*, SPC-2000045-MC, version 1.0. Software Productivity Consortium.

About the Systems and Software Consortium, Inc.

The Systems and Software Consortium, Inc. (SSCI) is a nonprofit partnership of market leaders, government agencies, and academic affiliates. As a consortium, SSCI enables industry and government to co-invest in the development of systems and software processes and capabilities that improve business performance. Members also have access to a team of technical experts whose collective knowledge of best practices and lessons learned gives SSCI the unique opportunity to offer practical advice and proven solutions.

The Consortium is interested in your comments and suggestions. Please send your thoughts and insights to ask-ssci@systemsandsoftware.org.

For more information about the Systems and Software Consortium, please visit www.systemsandsoftware.org