Automating Formal Specification-Based Testing To Appear: TAPSOFT '97

Michael R. Donat

University of British Columbia, 2366 Main Mall, Vancouver B.C. V6T 1Z4, Canada

Abstract. This paper presents a technique for automatically generating logical schemata that specify groups of black-box test cases from formal specifications containing universal and existential quantification. These schemata are called test frames. Previous automated techniques have dealt with languages based on propositional logic. Since this new technique deals with quantification it can be applied to more expressive specifications. This makes the technique applicable to specifications written at the system requirements level. The limitations imposed by quantification are discussed. Industrial needs are addressed by the capabilities of recognizing and augmenting existing test frames and by accommodating a range of specification-coverage schemes. The coverage scheme taxonomy introduced in this paper provides a standard for controlling the number of test frames produced. This technique is intended to automate portions of what is done manually by practitioners. Basing this technique on formal rules of logical derivation ensures that the test frames produced are logical consequences of the specification. It is expected that deriving test frames automatically will offset the cost of developing a formal specification. This tangible product makes formal specification more economically feasible for industry.

1 Introduction

The primary contribution of this paper is a technique for automatically transforming formal specifications containing universal and existential quantification into test frames which specify groups of black-box test cases. The second major contribution of this paper is a taxonomy for coverage schemes. This taxonomy provides a means of standardizing the number of tests to be performed on specific parts of the system. This is critical to industrial processes that must make appropriate trade-offs between available resources and the depth of testing required for a given part of the system.

Formal specifications based on mathematical semantics provide a basis for automatic test generation techniques. This mathematical structure allows formal specifications to be manipulated mechanically so that information contained within the specification can be isolated, transformed, assembled, and repackaged. In this manner, test frames for a system can be derived from its formal specification. The mathematical semantics of the specification language guarantee that the test frames are logical consequences of the specification.

Dick and Faivre [6], inspired by the work of Bernot, Gaudel, and Marre [3], showed how test cases could be generated automatically from unquantified predicate logic specifications using a specific coverage scheme. This form of logic is limited for general use in specifications at the system requirements level. Widely used languages such as Z [14] make use of quantification. The technique presented in this paper shows how to automatically generate test frames in the presence of quantified specifications using a variety of coverage schemes.

MacColl, Carrington, and Stocks [12] describe a mechanized but not automated approach to deriving test cases from formal specifications. They provide for a variety of *strategies* that could embody different coverage schemes.

Gaudel [9] describes a theory of testing based on algebraic specifications. These are different from the predicate logic specifications addressed in this paper. Algebraic specifications are characterized by the use of functions to denote operations. A set of axioms, typically expressed as universally quantified equations, defines a class of algebras. Each algebra is said to be a model of the specification. In contrast, predicate logic specifications typically use relations between states to denote operations and both universal and existential quantification are often present.

Despite these differences, similar concepts and problems arise when generating tests. The concepts defined by Gaudel, such as exhaustive test set, validity, unbias, selection and uniformity hypotheses, and the oracle problem, have counterparts within the context of predicate logic specifications. This paper contains only a brief description of the theory supporting the work presented here. A full discussion is given in [7]. In the context of either type of specification the number of tests produced must be controlled. This paper discusses a method of achieving the necessary control for boolean expressions using standardized coverage schemes.

Techniques of producing test case instances of test frames are part of a subsequent process and are not discussed here.

Section 2 sets the context that motivates this research. The notation used to present details of the technique is described in Section 3. Section 4 presents a general description of a process to generate test frames from formal specifications. This section also introduces and distinguishes the concepts of a specification, its test classes, the test frames that follow, and the test cases they describe. Section 5 details the test class algorithm. Test frames and how they are produced using various coverage schemes is discussed in Section 6.

2 Industrial Context

There are several different types of testing. Each type focuses on a different objective and a different abstract view of the software. Unit testing focuses on the robustness of individual components. Integration testing focuses on the correctness of the interfaces between components. This paper focuses on testing based on requirements specifications. An objective of this type of testing is to demonstrate to a customer or certification authority that the specified software has actually been built.

This testing is performed according to a set of test procedures. Each step in a test procedure is referred to as a test case. The purpose of each test case is to verify one or more requirements by the application of an external stimulus to the system and comparison of the actual response of the system against the expected response specified by the requirements. The analysis of requirements for the purpose of deriving tests at this level is generally limited to lexical analysis of the natural language text used to express the requirements.

This level of testing is "system level" in the sense that the internal structure of the system is not visible; all testing must be performed by means of external stimuli and observation of externally visible responses. It is "requirements-based" in contrast to other kinds of system level testing which, for instance, may be based on scenarios that attempt to approximate expected use of the system.

Test case derivation for large projects is typically a highly manual process. Teams of test engineers wade through large volumes of software specifications, interpret them to the best of their abilities, and from this generate appropriate suites of tests to apply to the developed systems. The process is very tedious and error prone, due to the possible ambiguities of natural language and the amount of detail involved. This intensity of labour coupled with the costs of ensuring test suite correctness provides a sizable economic motivation to automate as much of the test case derivation process as possible.

Toth and Joyce [15, 16] introduced the FORMATS Process as a way of applying formal methods to test case derivation. FORMATS is a two step process. Requirements specifications are formalized and type checked to ensure that they meet a certain level of correctness. Test cases are produced in the second step. Specifications are written in S [11], which is a typed predicate logic similar to that found in the HOL system [10]. S specifications are type checked using a tool called *Fuss*. To advance the ideas discussed in [16], the author has implemented a prototype test frame generator that employs the technique described in this paper.

There are four important issues in the FORMATS Process:

1. A range of coverage schemes may be employed depending on the amount of testing required.
2. Test suites should be as small as possible while still providing the desired coverage.
3. Specifications may change as the project progresses.
4. The test team may mandate specific tests.

When specification changes occur it is necessary to evaluate the impact this has on existing test suites previously constructed. Although generating a completely new test suite is possible, this is undesirable if testing has already begun. Performing a few new tests to augment positive results already obtained is less expensive than dismissing previous positive results and performing a larger number of different tests. As an example, consider the case where a portion of the specification is reworded for clarity or contractual reasons, but no implementation changes are necessary. If the test case generator produced new tests based on the rewording, unnecessary and perhaps costly testing would be done.

When particular tests are mandated, the test case generator must build a test suite around these given tests. This must be done in a manner that preserves the desired size and coverage for the test suite. Note that *coverage* refers to coverage of the specification and not code coverage of the implementation.

The technique presented here addresses each of these issues.

3 Notation

The technique presented in this paper is based on the logical relationships between elements within the specification. Since it is not tied to a particular specification language such as S or Z, standard logical expressions shall be used in the discussions that follow.

The following vocabulary will be helpful:

1. A *specification* of a system is a boolean expression relating the state of the system before the program executes to the state of the system after the program has executed. The expression is constructed from predicates, the logical connectives conjunction, disjunction, implication, and negation, ($\vee, \wedge, \Rightarrow$, and $\neg$), along with universal and existential quantification ($\forall$ and $\exists$).
2. An *atom* is either a predicate or a negated predicate.
3. A *stimulus* is an atom that only refers to the before state.
4. A *stimulus expression* is a boolean expression where each atom is a stimulus.
5. A *response* is an atom that contains at least one reference to the after state and may also refer to the before state, i.e. an atom that is not a stimulus.
6. A *response expression* is a boolean expression where each atom is a response.

A program specification can be of the form:

$$(S_1 \Rightarrow R_1) \wedge (S_2 \Rightarrow R_2) \wedge \ldots \tag{1}$$

where the S_i are stimulus expressions and the R_i are response expressions. This specifies a system that will satisfy R_i when given the stimulus S_i. In this specification, each implication describes a class of behaviour to be exhibited by the system.

To illustrate these definitions, consider the following example which is a modification[1] of an excerpt from Bernard's solution [2] to Abrial's steam boiler specification problem [1].

The specification problem is to formally specify requirements for a control system responsible for maintaining the correct level of water in a boiler attached to a steam driven turbine. One of the requirements is to identify whether or not any inconsistencies exist in the sensor readings.

[1] Modifications were made to construct a concise example and do not affect its logical complexity. The excerpt is similar to the VDM specification by Schinagl [13].

$$
\begin{array}{|l}
\hline
\textit{PHYSMESS} \\
\Delta WS \\
\hline
\neg\ OOTM' \Leftrightarrow \\
\quad (\exists_1\, n : \mathbb{N} \bullet Level\ n \in inmess) \land \\
\quad (\exists_1\, n : \mathbb{N} \bullet Steam\ n \in inmess) \land \\
\quad (\forall\, i : PUMP \bullet PumpState(i, TRUE) \in inmess \Leftrightarrow \\
\qquad \neg\ (PumpState(i, FALSE) \in inmess)) \land \\
\quad (\forall\, i : \mathbb{N} \bullet \exists\, b : bool \bullet PumpCtrState(i, b) \in inmess) \\
\hline
\end{array}
$$

The schema *PHYSMESS* sets the "out of order" indicator, *OOTM*, to true if and only if there is a detected malfunction. *inmess* is a set of input messages received from the sensors of the boiler system. *Level n* indicates the quantity of water in the boiler, *Steam n* indicates the quantity of steam coming from the boiler, *PumpState* indicates whether pump i is turned on or off, *PumpCtrState* indicates whether or not water is circulating from the pump to the boiler.

Expressed in predicate logic, *PHYMESS* is equivalent to:

$$
\begin{array}{l}
\neg OOTM' \Leftrightarrow \\
\ ((\exists! n.Level\ n \in inmess) \land \\
\ (\exists! n.Steam\ n \in inmess) \land \\
\ (\forall\, i.PumpState(i, T) \in inmess \Leftrightarrow \neg(PumpState(i, F) \in inmess)) \land \\
\ (\forall\, i.\exists\, b.(PumpCtrState(i, b) \in inmess)))
\end{array}
$$

Primed variables are references to the after state, thus $\neg OOTM'$ is a response. All the other atoms, such as $PumpState(i, TRUE) \in inmess$, are stimuli.

4 Process Overview

This section provides an overview of the test frame generation process.

Requirements specifications are written to be understood at particular levels of abstraction. For this reason, many details are hidden within definitions of more abstract concepts. Issues of clarity are left to the discretion of the specification authors. Hence, it must be assumed that the specification is an arbitrary logical expression.

Test classes are the intermediate step between the specification and test frames. A test class isolates one behaviour from the specification. The test class can be considered as a standard format for writing requirements. However, for practical reasons, it is unlikely that all specifications would be written as a simple conjunction of test classes.

A *test class* is an implication $S \Rightarrow R$, where S is a stimulus expression and R is a response expression. Quantifiers may appear anywhere in the test class and may also bind variables occurring in both S and R. The purpose of the test class is to isolate a class of behaviour based on the response. The first step of the test frame generation process is to transform the specification into its *test class normal form* such as (1) in Section 3. This is discussed in detail in Section 5.

Each test class is the ancestor of a set of test frames. A *test frame* is an implication $A \Rightarrow R$, where A is a conjunction of stimulus expressions and R is a response expression. Quantifiers may also bind variables occurring in both A and R. A test frame $A \Rightarrow R$ generated from the test class $S \Rightarrow R$ has the property that $A \Rightarrow S$. The generation of test frames is discussed in detail in Section 6.

The test frame generation process is as follows. Given a general specification E, a set of test classes $S_i \Rightarrow R_i$ are produced such that $E \Rightarrow (S_i \Rightarrow R_i)$. From each test class, a set of test frames $A_{ij} \Rightarrow R_i$ are produced such that $A_{ij} \Rightarrow S_i$. This ensures that each test frame is valid, i.e. $E \Rightarrow (A_{ij} \Rightarrow R_i)$.

A *test case* is an implication $t \Rightarrow R$, where t is a conjunction of atoms and R is a response expression. Quantifiers can only occur in R. Although it is desirable to derive test cases, these cannot, in general, be generated automatically from the type of specifications considered in this paper. However, much of the effort required to generate a test case can be performed automatically by producing a test frame.

Test data generation techniques, whether manual or machine assisted, can be applied to test frames to produce test cases $t_{ijk} \Rightarrow R_i^+$ such that $t_{ijk} \Rightarrow A_{ij}^+$ where $A_{ij}^+ \Rightarrow R_i^+$ is an instance of the (quantified) test frame $A_{ij} \Rightarrow R_i$. Discussion of these test data generation techniques is beyond the scope of the concept presented in this paper.

5 The Test Class Algorithm

The test class algorithm can be described as a function on boolean expressions. The result of applying this function to an expression, E, is a conjunction of test classes that is logically equivalent to E. The test class algorithm rewrites the specification into its test class normal form. This does not alter its logical content.

Assuming R is a response, S is a stimulus, T is the constant true, and F is the constant false, a definition for the test class function, TC, is:

$$\begin{array}{rcll}
TC(A \wedge B) &=& \mathit{RewriteAnd}(TC(A) \wedge TC(B)) & \text{conjunction} \\
TC(A \vee B) &=& \mathit{RewriteOr}(TC(A) \vee TC(B)) & \text{disjunction} \\
TC(\forall x.P) &=& \mathit{ForallIn}(\forall x.TC(P)) & \text{quantification} \\
TC(\exists x.P) &=& \mathit{ExistsIn}(\exists x.TC(P)) & \text{quantification} \\
TC(A \Rightarrow B) &=& TC(\neg A \vee B) & \text{implication} \\
TC(R) &=& T \Rightarrow R & \text{response} \\
TC(S) &=& \neg S \Rightarrow F & \text{stimulus}
\end{array}$$

Negated expressions are dealt with by applying DeMorgan's laws to move the negation inwards and proceeding.

The function *RewriteAnd* combines like antecedents and consequents using the equivalences

$$\forall A, B, C.(A \Rightarrow B) \wedge (A \Rightarrow C) = A \Rightarrow (B \wedge C)$$

$$\forall A, B, C.(A \Rightarrow C) \wedge (B \Rightarrow C) = (A \vee B) \Rightarrow C \ .$$

The function *RewriteOr* first reduces any AND/OR connectives above the test classes from $TC(A)$ and $TC(B)$ to conjunctive normal form. Next, any universal quantifiers are pulled from $TC(A)$ and $TC(B)$ so they are outside the disjunctions. This is done using the equivalences

$$\begin{aligned} &\forall P, Q.(\forall x.Q) \vee P = \forall x.Q \vee P \\ &\forall P, Q.P \vee (\forall x.Q) = \forall x.P \vee Q \ , \end{aligned}$$

where x is alpha converted if necessary to avoid capturing any free occurrence of x in P. Finally, the test classes are OR'd together using the equivalence

$$\forall S_1, S_2, R_1, R_2.(S_1 \Rightarrow R_1) \vee (S_2 \Rightarrow R_2) = S_1 \wedge S_2 \Rightarrow R_1 \vee R_2 \ .$$

The function *ForallIn* moves the universal quantifier into the conjunction of test classes produced by $TC(P)$ using the equivalences

$$\begin{aligned} &\forall P, Q.(\forall x.P \Rightarrow Q) = (\exists x.P) \Rightarrow Q \\ &\forall P, Q.(\forall x.Q \Rightarrow P) = Q \Rightarrow (\forall x.P) \\ &\forall P, Q.(\forall x.P \wedge Q) = (\forall x.P) \wedge Q \\ &\forall P, Q.(\forall x.Q \wedge P) = Q \wedge (\forall x.P) \\ &\forall M, P.(\forall x.M \wedge P) = (\forall x.M) \wedge (\forall x.P) \ , \end{aligned}$$

where x is free in P and M, and x is not free in Q.

The function *ExistsIn* moves the existential quantifier into the test class using the equivalences

$$\begin{aligned} &\forall P, Q.(\exists x.P \Rightarrow Q) = (\forall x.P) \Rightarrow Q \\ &\forall P, Q.(\exists x.Q \Rightarrow P) = Q \Rightarrow (\exists x.P) \\ &\forall M, P.(\exists x.M \Rightarrow P) = (\forall x.M) \Rightarrow (\exists x.P) \ , \end{aligned}$$

where x is free in P and M, and x is not free in Q.

Quantification does impose certain limitations on the test class algorithm. However, specifications exercising these limits may be deemed too weak. Note that *ForallIn* will not be successful in moving the universal quantifier into the conjunction if there is an existential quantifier in the way,

$$\text{e.g. } \forall x.\exists y.(S_1 \Rightarrow R_1) \wedge (S_2 \Rightarrow R_2) \ . \tag{2}$$

Similarly, *ExistsIn* will not be successful in moving the existential quantifier into a test class if $TC(P)$ produces more than one test class as in (2), or if the single test class has a universal quantifier,

$$\text{e.g. } \exists x.\forall y.(S_1 \Rightarrow R_1) \ . \tag{3}$$

It could be argued that test class (3) can be dismissed as being too weak to be a reasonable requirement. A similar argument could be made against the test classes in (2).

5.1 Example

In our example, $\Leftrightarrow$ is defined as $\forall\, A, B.(A \Leftrightarrow B) = (A \Rightarrow B) \wedge (B \Rightarrow A)$ and $\exists! x.P\,x$ is defined as $(\exists\, x.P\,x) \wedge (\forall\, x, y.P\,x \wedge P\,y \Rightarrow (x = y))$. Applying the TC algorithm begins with the conjunction rule:

$$
\begin{array}{l}
\quad TC(\neg OOTM' \Leftrightarrow \\
\quad\;((\exists\, n.Level\ n \in inmess) \wedge \\
\quad\;(\forall\, n, m.(Level\ n \in inmess) \wedge (Level\ \ m \in inmess) \Rightarrow (n = m)) \wedge \\
\quad\;(\exists\, n.Steam\ n \in inmess) \wedge \\
\quad\;(\forall\, n, m.(Steam\ n \in inmess) \wedge (Steam\ \ m \in inmess) \Rightarrow (n = m)) \wedge \\
\quad\;(\forall\, i.PumpState(i, T) \in inmess \Leftrightarrow \neg PumpState(i, F) \in inmess) \wedge \\
\quad\;(\forall\, i.\,\exists\, b.PumpCtrState(i, b) \in inmess))) \\
= RewriteAnd(TC(\neg OOTM' \Rightarrow \\
\quad\;((\exists\, n.Level\ n \in inmess) \wedge \\
\quad\;(\forall\, n, m.(Level\ n \in inmess) \wedge (Level\ \ m \in inmess) \Rightarrow (n = m)) \wedge \\
\quad\;(\exists\, n.Steam\ n \in inmess) \wedge \\
\quad\;(\forall\, n, m.(Steam\ n \in inmess) \wedge (Steam\ \ m \in inmess) \Rightarrow (n = m)) \wedge \\
\quad\;(\forall\, i.PumpState(i, T) \in inmess \Leftrightarrow \neg PumpState(i, F) \in inmess) \wedge \\
\quad\;(\forall\, i.\,\exists\, b.PumpCtrState(i, b) \in inmess))) \wedge TC(\ldots))
\end{array}
$$

The next operation is to rewrite the implication of the first TC term and use the rule for disjunction:

$$= RewriteAnd(RewriteOr(TC(\neg\neg OOTM') \vee TC(\ldots)) \wedge TC(\ldots))$$

The double negation is removed and the response rule is then applied:

$$= RewriteAnd(RewriteOr((T \Rightarrow OOTM') \vee TC(\ldots)) \wedge TC(\ldots))$$

Using the rule for conjunction on the next TC term produces:

$$
\begin{array}{l}
= RewriteAnd(RewriteOr((T \Rightarrow OOTM') \vee \\
\quad RewriteAnd(TC(\exists\, n.Level\ n \in inmess) \wedge TC(\ldots)) \wedge TC(\ldots)))
\end{array}
$$

The quantification rule followed by the stimulus rule gives:

$$
\begin{array}{l}
= RewriteAnd(RewriteOr((T \Rightarrow OOTM') \vee \\
\quad RewriteAnd(ExistsIn(\exists\, n.\neg(Level\ n \in inmess) \Rightarrow F) \wedge TC(\ldots)) \\
\;\wedge\, TC(\ldots)))
\end{array}
$$

Applying $ExistsIn$ gives:

$$
\begin{array}{l}
= RewriteAnd(RewriteOr((T \Rightarrow OOTM') \vee \\
\quad RewriteAnd(((\forall\, n.\neg(Level\ n \in inmess)) \Rightarrow F) \wedge TC(\ldots)) \wedge TC(\ldots)))
\end{array}
$$

A full application of the algorithm to the next TC term produces:

$$
\begin{aligned}
&= RewriteAnd(RewriteOr((T \Rightarrow OOTM') \vee \\
&\quad RewriteAnd(((\forall\, n. \neg(Level\ n \in inmess)) \Rightarrow F) \wedge \\
&\quad\ (((\exists\, n, m.(Level\ n \in inmess) \wedge (Level\ n \in inmess) \wedge \neg(n = m)) \vee \\
&\quad\ (\forall\, n. \neg(Steam\ n \in inmess)) \vee \\
&\quad\ (\exists\, n, m.(Steam\ n \in inmess) \wedge (Steam\ n \in inmess) \wedge \neg(n = m)) \vee \\
&\quad\ (\exists\, i.(PumpState(i, T) \in inmess \wedge PumpState(i, F) \in inmess) \vee \\
&\quad\quad (\neg(PumpState(i, T) \in inmess) \wedge \neg(PumpState(i, F) \in inmess))) \vee \\
&\quad\ (\exists\, i. \forall\, b. \neg(PumpCtrState(i, b) \in inmess))) \\
&\quad\ \Rightarrow F)) \wedge \\
&\quad TC(\ldots)))
\end{aligned}
$$

Since the consequents of the two inner-most implications are identical (F), applying the inner-most $RewriteAnd$ produces:

$$
\begin{aligned}
&= RewriteAnd(RewriteOr((T \Rightarrow OOTM') \vee \\
&\quad (((\forall\, n. \neg(Level\ n \in inmess)) \vee \\
&\quad (\exists\, n, m.(Level\ n \in inmess) \wedge (Level\ n \in inmess) \wedge \neg(n = m)) \vee \\
&\quad (\forall\, n. \neg(Steam\ n \in inmess)) \vee \\
&\quad (\exists\, n, m.(Steam\ n \in inmess) \wedge (Steam\ n \in inmess) \wedge \neg(n = m)) \vee \\
&\quad (\exists\, i.(PumpState(i, T) \in inmess \wedge PumpState(i, F) \in inmess) \vee \\
&\quad\ (\neg(PumpState(i, T) \in inmess) \wedge \neg(PumpState(i, F) \in inmess))) \vee \\
&\quad (\exists\, i. \forall\, b. \neg(PumpCtrState(i, b) \in inmess))) \\
&\quad \Rightarrow F) \wedge \\
&\quad TC(\ldots)))
\end{aligned}
$$

Applying $RewriteOr$ combines the response and stimuli to produce the first test class:

$$
\begin{aligned}
&= RewriteAnd(\\
&\quad (((\forall\, n. \neg(Level\ n \in inmess)) \vee \\
&\quad (\exists\, n, m.(Level\ n \in inmess) \wedge (Level\ n \in inmess) \wedge \neg(n = m)) \vee \\
&\quad (\forall\, n. \neg(Steam\ n \in inmess)) \vee \\
&\quad (\exists\, n, m.(Steam\ n \in inmess) \wedge (Steam\ n \in inmess) \wedge \neg(n = m)) \vee \\
&\quad (\exists\, i.(PumpState(i, T) \in inmess \wedge PumpState(i, F) \in inmess) \vee \\
&\quad\ (\neg(PumpState(i, T) \in inmess) \wedge \neg(PumpState(i, F) \in inmess))) \vee \\
&\quad (\exists\, i. \forall\, b. \neg(PumpCtrState(i, b) \in inmess))) \\
&\quad \Rightarrow OOTM') \wedge \\
&\quad TC(\ldots)))
\end{aligned}
$$

Continuing with the remaining TC term produces the second test class:

$$\begin{aligned}
&(\exists\, n.\mathit{Level}\ n \in \mathit{inmess}) \wedge \\
&(\forall\, n, m.\neg(\mathit{Level}\ n \in \mathit{inmess}) \vee \neg(\mathit{Level}\ m \in \mathit{inmess}) \vee (n = m)) \wedge \\
&(\exists\, n.\mathit{Steam}\ n \in \mathit{inmess}) \wedge \\
&(\forall\, n, m.\neg(\mathit{Steam}\ n \in \mathit{inmess}) \vee \neg(\mathit{Steam}\ m \in \mathit{inmess}) \vee (n = m)) \wedge \\
&(\forall\, i.(\neg(\mathit{PumpState}(i, T) \in \mathit{inmess}) \vee \neg(\mathit{PumpState}(i, F) \in \mathit{inmess})) \wedge \\
&\ (\mathit{PumpState}(i, T) \in \mathit{inmess} \vee \mathit{PumpState}(i, F) \in \mathit{inmess}) \wedge \\
&(\forall\, i.\, \exists\, b.\mathit{PumpCtrState}(i, b) \in \mathit{inmess}) \\
&\Rightarrow \neg \mathit{OOTM}' \ .
\end{aligned}$$

6 Generating Test Frames

As defined previously, a test frame from a given test class $S \Rightarrow R$ is an implication $A \Rightarrow R$, where $A \Rightarrow S$, A is a conjunction of stimulus expressions, and R is a response expression. Quantifiers may also bind variables occurring in both A and R.

A variety of different test frame sets can be constructed from a test class. One possible set of test frames is the one derived from a disjunctive normal form (DNF) of the test class antecedent. In the context of our industrial process, this presents a problem. If an existing test suite contains a valid test frame that does not correspond to a term in the DNF of the antecedent of the test class, it will not be recognized as valid and will be replaced. This is not desirable since we wish to replace tests only when necessary. This situation can occur when the test class antecedent represents a function having more than one DNF.[2]

Recognizing valid test frames in an existing test suite and then constructing other test frames around them is an NP-complete problem [8]. The binary decision diagram (BDD) [4] is a convenient tool for addressing this issue. The technique described here uses BDDs to perform test frame recognition, construction, and selection. The strategy for generating test frame antecedents is:

1. Generate the set of prime implicants[3] for the antecedent of the test class.
2. Identify any existing or mandated valid test frames.
3. Augment this set with other elements from the set of prime implicants to construct a set with the desired specification coverage properties.

6.1 Constructing the BDD

BDDs encode unquantified boolean expressions. Quantifiers within the test class place a limit on the granularity of the terms that appear in test frames. To

[2] Consider the function $(a \wedge \neg c) \vee (\neg b \wedge c) \vee (\neg a \wedge b)$ and its alter ego $(a \wedge \neg b) \vee (\neg a \wedge c) \vee (b \wedge \neg c)$.

[3] An implicant of a formula is a conjunction of variables that imply the formula. An implicant is prime if there is no other implicant that implies it.

obtain an unquantified expression from the test class antecedent, quantifiers are pushed inwards to group the quantifiers as tightly as possible to the stimuli they quantify. Existential quantifiers that are not blocked by universal quantifiers are then moved outside the implication where they become universal quantifiers. This minimizes the number of quantifiers in the test class antecedent.

As an example, consider

$$\begin{aligned}
&(\forall\, x.\, \exists\, y. A(x) \wedge (B \vee C(y))) \Rightarrow R \\
&= ((\forall\, x. A(x)) \wedge (B \vee \exists\, y. C(y))) \Rightarrow R \\
&= (\exists\, y. (\forall\, x. A(x)) \wedge (B \vee C(y))) \Rightarrow R \\
&= \forall\, y. ((\forall\, x. A(x)) \wedge (B \vee C(y))) \Rightarrow R \ .
\end{aligned}$$

Applying this process to the steam boiler test classes results in:

$$\begin{aligned}
&\forall\, n, m, i. \\
&\quad (\forall\, n. \neg(\mathit{Level}\ n \in \mathit{inmess})) \vee \\
&\quad ((\mathit{Level}\ n \in \mathit{inmess}) \wedge (\mathit{Level}\ m \in \mathit{inmess}) \wedge \neg(n = m)) \vee \\
&\quad (\forall\, n. \neg(\mathit{Steam}\ n \in \mathit{inmess})) \vee \\
&\quad ((\mathit{Steam}\ n \in \mathit{inmess}) \wedge (\mathit{Steam}\ m \in \mathit{inmess}) \wedge \neg(n = m)) \vee \\
&\quad ((\mathit{PumpState}(i, T) \in \mathit{inmess} \wedge \mathit{PumpState}(i, F) \in \mathit{inmess}) \vee \\
&\quad\ \ (\neg(\mathit{PumpState}(i, T) \in \mathit{inmess}) \wedge \neg(\mathit{PumpState}(i, F) \in \mathit{inmess}))) \vee \\
&\quad (\forall\, b. \neg(\mathit{PumpCtrState}(i, b) \in \mathit{inmess})) \\
&\quad \Rightarrow \mathit{OOTM}'
\end{aligned}$$

$$\begin{aligned}
&\forall\, n_1, n_2. \\
&\quad (\mathit{Level}\ n_1 \in \mathit{inmess}) \wedge \\
&\quad (\forall\, n, m. \neg(\mathit{Level}\ n \in \mathit{inmess}) \vee \neg(\mathit{Level}\ m \in \mathit{inmess}) \vee (n = m)) \wedge \\
&\quad (\mathit{Steam}\ n_2 \in \mathit{inmess}) \wedge \\
&\quad (\forall\, n, m. \neg(\mathit{Steam}\ n \in \mathit{inmess}) \vee \neg(\mathit{Steam}\ m \in \mathit{inmess}) \vee (n = m)) \wedge \\
&\quad (\forall\, i. (\neg(\mathit{PumpState}(i, T) \in \mathit{inmess}) \vee \neg(\mathit{PumpState}(i, F) \in \mathit{inmess})) \wedge \\
&\quad\ \ (\mathit{PumpState}(i, T) \in \mathit{inmess} \vee \mathit{PumpState}(i, F) \in \mathit{inmess})) \wedge \\
&\quad (\forall\, i. \exists\, b. \mathit{PumpCtrState}(i, b) \in \mathit{inmess}) \\
&\quad \Rightarrow \neg \mathit{OOTM}' \ .
\end{aligned}$$

A BDD representation is constructed by substituting a variable for each quantified subexpression and unquantified stimulus. The expressions and stimuli represented by BDD variables are referred to as *frame stimuli*.

The antecedent of the first test class (above) can be represented with the unquantified expression:

$$V_1 \vee (V_2 \wedge V_3 \wedge \neg E) \vee W_1 \vee (W_2 \wedge W_3 \wedge \neg E) \vee ((X \wedge Y) \vee (\neg X \wedge \neg Y)) \vee Z \quad (4)$$

where

$V_1 = \forall\, n . \neg(Level\ n \in inmess)$ $\qquad W_1 = \forall\, n . \neg(Steam\ n \in inmess)$
$V_2 = Level\ n \in inmess$ $\qquad W_2 = Steam\ n \in inmess$
$V_3 = Level\ m \in inmess$ $\qquad W_3 = Steam\ m \in inmess$

$X = PumpState(i, T) \in inmess$ $\qquad Z = \forall\, b . \neg(PumpCtrState(i, b) \in inmess)$
$Y = PumpState(i, F) \in inmess$ $\qquad E = (n = m)$

The set of prime implicants is then generated from the BDD representation of this expression. The corresponding test frames are:

$$(\forall\, n . \neg(Level\ n \in inmess)) \Rightarrow OOTM' \qquad (\forall\, n . \neg(Steam\ n \in inmess)) \Rightarrow OOTM'$$

$$\forall\, n, m . Level\ n \in inmess \wedge Level\ m \in inmess\ \wedge \neg(n = m) \Rightarrow OOTM' \qquad \forall\, n, m . Steam\ n \in inmess \wedge Steam\ m \in inmess\ \wedge \neg(n = m) \Rightarrow OOTM'$$

$$\forall\, i . PumpState(i, T) \in inmess \wedge PumpState(i, F) \in inmess \Rightarrow OOTM' \qquad \forall\, i . \neg(PumpState(i, T) \in inmess) \wedge \neg(PumpState(i, F) \in inmess) \Rightarrow OOTM'$$

$$\forall\, i . (\forall\, b . \neg(PumpCtrState(i, b) \in inmess)) \Rightarrow OOTM' \ .$$

Since the antecedent of the second test class is a conjunction of frame stimuli, there is only one test frame; the one identical to the test class.

Although quantifiers were used liberally throughout the specification, reasonable test frames could still be generated automatically. Any manual test case generation that remains is less tedious and less error prone than it would have been without being able to use the test frames as a starting point.

6.2 Coverage Criteria

With the set of prime implicants at hand, several coverage schemes can be defined. These can then be used at the discretion of the practitioner. The test frame generation technique places no restrictions on the coverage scheme.

The author proposes the following taxonomy for coverage schemes:

1. **All points:** This is the DNF of Dick and Faivre where each test frame specifies the truth or falsehood of each of the frame stimuli from the test class stimulus expression.
2. **Implicant:** Test frames are produced for each prime implicant.
3. **DNF:** Test frames are produced for a subset of prime implicants whose disjunction corresponds to a DNF of the test class stimulus expression.
4. **Partition:** A subset of prime implicants are used to determine an implicant set that is similar to DNF coverage, but the implicants are pair-wise contradictory. i.e. There is no test case that will satisfy any two test frames.

5. **Term:** Test frames are produced for a subset of prime implicants such that each frame stimuli from the test class stimulus expression is present in at least one of the selected prime implicants.

The differences between these coverage schemes can be illustrated by considering the number of terms produced when applied to the expression in Figure 1. This figure shows the points where the expression is true and compares the Karnaugh maps of the coverage schemes defined above. Each bubble represents the antecedent of a test frame. The coverage schemes produce 8, 5, 4, 4, and 3 test frames, respectively.

Comparison of coverage schemes applied to
$(\neg W \wedge \neg X \wedge \neg Y) \vee (\neg W \wedge \neg Y \wedge Z) \vee (X \wedge Y) \vee (W \wedge Y \wedge \neg Z)$

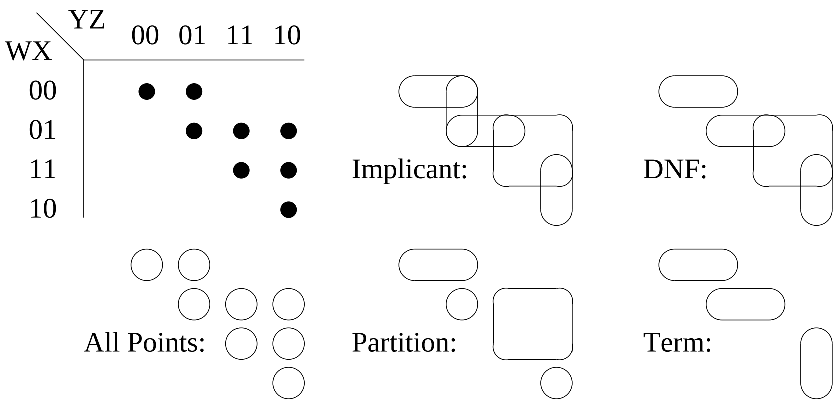

Fig. 1.

Term coverage is of interest since it is linear with respect to the size of the specification rather than exponential, as are the others. Note that term coverage does not produce test frames that cover two of the eight all-points cases, $W \wedge X \wedge Y \wedge Z$ and $\neg W \wedge X \wedge Y \wedge \neg Z$. This is the compromise made in order to produce fewer tests.

7 Conclusions

The technique described in this paper addresses the process of deriving test frames from formal requirements specifications. A prototype has been constructed that demonstrates that this process can be automated for specifications written in a predicate logic with universal and existential quantification. Augmenting existing test suites will be implemented in the near future.

As noted by Gaudel, predicate logic specifications are more general than algebraic specifications. However, the price of this generality is the restriction that, in general, only test frames can be generated automatically. Algebraic techniques such as [3] can generate test data corresponding to what this paper refers to as test cases.

The automatic construction of a state machine to facilitate test case sequencing is not considered here. For requirements specifications, specifying the state machine explicitly may be more appropriate, as in Büssow and Webers' hybrid Statecharts-Z approach [5].

BDDs provide a valuable and powerful mechanism for recognizing existing test frames. This same approach should also be able to match white-box test data to the corresponding test frames, provided that a mapping from the white-box vocabulary to that of the specification is given. This would provide a mechanism for generating oracles for white-box tests.

Quantifiers place limits on the depth to which automation can go in producing test frames. Further research is needed to assess the impact of quantified expressions within test frames and the frequency with which they typically occur. With respect to the limits existential quantification places on generating test classes, further research will be needed to determine if this limitation is significant.

In spite of these limitations, the fact that these components are identified by the technique and automatically carried through to test frames constitutes a large savings in manual effort. The effort saved is the effort to generate the test frames manually along with the effort required to ensure they were generated correctly.

The use of prime implicants ensures that existing valid test frames or mandated tests stated in terms of test frames will be recognized. This represents a savings of testing effort and provides flexibility. The use of prime implicants also provides a mechanism by which the coverage scheme can be parameterized.

The information necessary for producing oracles for the test frames is produced at the time the test class is generated. The oracle is represented by the consequent of the test class. However, such oracles must be used with caution. As Gaudel points out, implementing such oracles relies on the correctness of the implementation of the oracle function.

8 Acknowledgements

This work is partially funded by the British Columbia Advanced Systems Institute and Hughes Aircraft of Canada, Limited. The author wishes to thank the reviewers for their comments and direction towards additional important related work.

References

1. Jean-Raymond Abrial. Steam boiler control specification problem. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods*

for Industrial Applications: Specifying and Programming the Steam Boiler Control, volume 1165 of *Lecture Notes in Computer Science*, pages 500–509, October 1996. http://www.informatik.uni-kiel.de/˜procos/dag9523/dag9523.html.

2. Pascal Bernard. A Z specification of the boiler. http://www.informatik.uni-kiel.-de/˜procos/dag9523/bernard-fulltext.ps.Z, January 1996.
3. G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications. *Software Engineering Journal*, 6(6), November 1991.
4. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. Robert Büssow and Matthias Weber. A steam-boiler control specification with statecharts and Z. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 109–128, October 1996.
6. Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe '93*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag, 1993.
7. Michael R. Donat. *Automating System-level Testing Based on Quantified Formal Specifications*. PhD thesis, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada, 1997. In preparation.
8. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H.Freeman and Company, San Francisco, 1979.
9. Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT: 6th International Joint Conference on Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96, 1995.
10. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
11. Jeffrey J. Joyce, Nancy Day, and Michael R. Donat. S: A machine readable specification notation based on higher order logic. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 285–299. Springer-Verlag, 1994.
12. Ian MacColl, David Carrington, and Philip Stocks. An experiment in specification-based testing. In K. Ramamohanarao, editor, *19th Australasian Computer Science Conference Proceedings (ACSC'96)*, pages 159–168, 1996.
13. Christian P. Schinagl. VDM specification of the steam-boiler control using RSL notation. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 428–452, October 1996.
14. J. Michael Spivey. *Understanding Z: A Specification language and its formal semantics*. Cambridge University Press, 1988.
15. K. Toth and J. Joyce. Industrialization of formal methods through process definition. In *5th Annual Symposium of the National Council on Systems Engineering*, Boston, July 1995. National Council on Systems Engineering. http://www.incose.org/.
16. Kalman Toth, Michael R. Donat, and Jeffrey J. Joyce. Generating test cases from formal specifications. In *6th Annual Symposium of the International Council on*

Systems Engineering, Boston, July 1996. International Council on Systems Engineering. http://www.incose.org/.