Test Oracles

Luciano Baresi
Dip. Elettronica e Informazione
Politecnico di Milano
Milano, Italy
baresi@elet.polimi.it

Michal Young
Dept. of Computer Science
University of Oregon
Eugene, Oregon, USA
michal@cs.uoregon.edu

4th August 2001

Abstract

All software testing methods depend on the availability of an *oracle,* that is, some method for checking whether the system under test has behaved correctly on a particular execution. An ideal oracle would provide an unerring pass/fail judgment for any possible program execution, judged against a natural specification of intended behavior. Practical approaches must make compromises to balance trade-offs and provide useful capabilities. This report surveys proposed approaches to the oracle problem that are general in the sense that they require neither pre-computed input/output pairs nor a previous version of the system under test. The survey is not encyclopedic, but discusses representative examples of the main approaches and tactics for solving common problems.

Partially supported by the Italian National Research Council (CNR). This work has also been supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Contents

1 Introduction

All software testing methods depend on the availability of an *oracle*, that is, some method for checking whether the system under test has behaved correctly on a particular execution. In much of the research literature on software test case generation or test set adequacy, the availability of oracles is either explicitly or tacitly assumed, but applicable oracles are not described. In the current industrial practice of software testing, the oracle is often a human being. Relying on a human to assess program behaviors has two evident drawbacks: accuracy and cost. While the human "eyeball oracle" has an advantage over more technical means in interpreting incomplete, natural-language specifications, humans are prone to error when assessing complex behaviors or detailed, precise specifications, and the accuracy of the eyeball oracle drops precipitously with increases in the number of test runs to be evaluated. Even if it were more dependable, the eyeball oracle is prohibitively expensive for large volumes of test cases, and so may become a limiting factor when other parts of testing are accelerated with automation.

An ideal test oracle would satisfy desirable properties of program specifications, such as being complete but avoiding over-specification, while also being efficiently checkable. These properties are in conflict, and many of the interesting issues and trade-offs in the design of test oracle systems come in various ways that tensions between desirable properties of specifications and necessary properties of implementations are resolved. As an oracle system takes on more of the capabilities of a "real" specification language, or provides more powerful facilities for deriving run-time checks from external specifications, several problems must be solved. Approaches to bridging the gap usually involve some combination of restricting the specification language to what can be effectively or efficiently checked (e.g., disallowing quantification over infinite sets), mapping implementation entities to specification-level entities, and/or taking advantage of the peculiarities of particular application domains.

The research literature on test oracles is a relatively small part of the research literature on software testing. Some older proposals base their analysis either on the availability of pre-computed input/output pairs [Pan78, Ham77] or on a previous version of the same program, which is presumed to be correct [Cha82]. The former hypothesis is usually too simplistic: being able to derive a significant set of input/output pairs would imply the capability of analyzing the system outcome. The latter hypothesis sometimes applies to regression testing, but is not sufficient in the general case. Weyuker has set forth some of the basic problems and argued that truly general test oracles are often unobtainable [Wey82].

This report surveys proposed approaches to automated test oracles that are general in the sense that they require neither pre-computed input/output pairs nor a previous version of the system under test. The survey is thematic rather than chronological, grouping systems to compare and contrast related approaches to variants of a few basic problems and design trade-offs.

2 Oracles for Transducers

Many programs are transducers that read an input sequence and produce an output sequence, maintaining a logical correspondence between the input and output structures. For example, a very large number of programs in web services are transducers from some native file format to hypertext markup language (HTML). It is not easy to express the intended behavior of these transducers in internal assertions or interface specifications for program modules; it is preferable to express and check the relation between the input sequence and output sequence. A specification for such a program, and a test oracle derived from that specification, must be based on a description of those structures.

The primary technology for describing and recognizing logical structures in textual input is parsing with context-free grammars, so it should not be surprising that grammars would play a part in specifying and checking transducers. Day and Gannon [DG85] have described a system that translates a formal specification of input and output files into an automated oracle. The prototype system described by Day and Gannon is specific to programs written in CF Pascal, a simplified version of Pascal with only `char` and `text` as primitive data types, but in principle it should be applicable to other languages including the scripting languages (Perl, Awk, Python, et al) commonly used to write simple transducers.

The specifications from which Day and Gannon extract test oracles are divided into a syntax section and a semantics section. The syntax uses two BNF grammars (see Example 1) to specify the format of input and output files, respectively, at the character level. The semantics defines rules (see Example 2) that specify the relationship the output must have with the input.

Example 1 *An example specification taken from [DG85], which requires a file (line) t of text and a file (line) b of blanks, as input, and produces a file (line) as output that contains t right justified to the length of b. Blanks that separate words should be distributed equally.*

```
FILES:
FileIn, FileOut, Width;

SYNTAX:
FILE = Width;
EOLN_TOKEN = ON;

<Width>        ::= <blank_string> <eoln>;
<blank_string> ::= <blank> <blank_string> |
                   <blank>;

FILE = FileIn;
EOLN_TOKEN = ON;

<FileIn>       ::= <Wlist> <blank_string> <eoln> |
                   <blank_string> <eoln>;
<Wlist>        ::= <Wlist> <blank> <word> | <word>;
<word>         ::= <word> <char> | <char>;
<blank_string> ::= <blank> <blank_string> | <lambda>;
```

```
FILE = FileOut;
EOLN_TOKEN = ON;

<FileOut>      ::= <Wblist> <eoln> |
                   <Wblist> <blank_string> <eoln> |
                   <blank_string> <eoln>;
<Wblist>       ::= <Wblist> <blank_string> <word> |
                   <word>;
<word>         ::= <word> <char> | <char>;
<blank_string> ::= <blank> <blank_string> | <blank>;
```

The first set of rules defines file `Width`*: It can be a sequence of blanks, possibly empty. The second set of rules defines file* `fileIn` *as a sequence of words separated by blanks (*`lambda` *is the empty token). The third set of rules similarly defines file* `fileOut`.

□

Semantic rules can be composed of both the standard comparisons between integers ($=$, $<>$, $<$, $<=$, $>$, and $>=$), booleans ($=$ and $<>$), and lists, bags, and sets ($=$ and $<>$), and user-defined functions. Special-purpose functions are provided for operating on textual sequences: `Chars` and `Words` decompose a file into lines, each containing a single character or word, respectively. `List`, `Set`, and `Bag` create lists, sets, and bags from files that contain components on different lines. `Number` returns the number of elements (lines) in a file.

The user-defined functions are written in the implementation language of the program under test, in this case Pascal. The only input must be a text file, while the output can be another text file, a boolean, or an integer.

Example 2 *The semantic rules for the previous BNF grammars.*

```
FUNCTIONS:
BString_Lengths: File;
EQ_Distw: Boolean;

SEMANTICS:
Number(Chars(FileIn)) <= Number(Chars(Width))
List(Words(FileOut)) = List(Words(FileIn))
Number(Chars(FileOut)) = Number(Chars(Width))
EQ_Dist(BString_Lengths(FileOut)) = TRUE
```

This section comprises two special-purpose functions and four rules. Function `BString_Lengths` *returns a file that contains the lengths of blank strings separating the words on a file; function* `EQ_Dist` *returns TRUE if the lengths are equally distributed, that is, the difference between the greatest and lowest values is one.*

The first rule requires the length of the input file be less than or equal to the length of the file of blanks (`Width`*.) The second rule states that he lists of words in the input and output files must be the same. The third rule requires that the number of characters in files* `FileOut` *and* `Width` *be the same. The forth rule requires the blanks to be equally distributed.*

□

The syntax and semantics sections are compiled together to obtain an oracle program for checking consistency of an output text with the corresponding input text.

3 Embedded Assertion Languages

Assertion languages allow expressions of intent to be embedded directly in program source code. Typical embedded assertion languages state properties to be checked at a particular control point in the program, directly in terms of programming language constructs and entities.[1] The prototypical embedded assertion language is the `assert` macro of the C programming language, which simply evaluates a boolean expression and prints an error message if the expression does not evaluate to *true*.

Early development of embedded assertion languages was aimed at least as much toward debugging as testing, with no particular emphasis on relating embedded assertions to a more abstract or global specification of intent. *IVTS* [Tay83] and *Anna* [Lv85] are among the first assertion systems in which embedded assertions are considered as a form of program specification in their own right. Considering embedded assertions as specifications, and assertion support as a way of using those specifications as test oracles, raises several problems, among them:

Non-local assertions: Embedded assertions are evaluated at particular points in the execution of a program, typically by treating the assertion as a program statement. Specifications sometimes describe properties that should be invariant during a computation, independent of control point, but it would be unwieldy to place corresponding assertions at all relevant program points. Many embedded assertion languages provide for precondition / postcondition pairs associated with a procedure as a whole to be evaluated at the beginning and return(s) from the procedure. Class invariants, supported by assertion languages for several object-oriented languages, are essentially post-conditions associated with the class constructor and each method (or each method that modifies the object state). Much less common is support for assertions that are evaluated at each point where a constraint expressed in an assertion could be violated.[2]

State caching: Specifications often constrain relations between values at different points in execution. In particular, procedure postcondition assertions typically relate the program state before and after execution of the procedure. Evaluation of such assertions requires saving a copy of parts or all of the "before" values mentioned in the assertion. This can be problematic when some of those values are large or complex, such as when the procedure under test manipulates a linked data structure.

Auxiliary variables: In addition to "before" values, program specifications may refer to other entities that do not exist in normal program evaluation. These are known as "ghost" or "auxiliary" variables. Several assertion languages provide a means

[1] We will treat separately systems in which assertions are used primarily to associate program state with an external model or specification. We also do not consider a notation to be an "embedded assertion language" if it is largely distinct from the underlying programming language, regardless of whether it is embedded in the programming language.

[2] If an assertion applies to each instance of a class then a class invariant suffices for evaluating the assertion at each point where it could be violated. Class invariants do not provide this functionality when the assertion should apply only to particular object instances, or the assertion applies to something outside the class system, e.g., an `int` variable in Java.

to define and use auxiliary variables. A general rule is that normal program execution must not be affected by any computation on auxiliary variables.

Quantification: Specifications make heavy use of universal and existential quantification. Sometimes evaluation of quantifiers is straightforwardly mapped to program loops, but this is not always an acceptable strategy. In a specification language designed for describing required program behavior as clearly and succinctly as possible, it is natural to make free use of quantification over large and even over infinite sets. For example, it is perfectly reasonable to state that all the elements of array A occur in array B as

$$\forall_{i \in 0..length(A)} \exists_{j \in 0..length(B)} : \text{A[i] = B[j]}$$

The loop interpretation of the quantifiers is problematic if A and B each have 1000 items, particularly if the assertion expresses an invariant that is preserved by swaps of two elements of A within a tight loop. Worse, there is nothing in principle wrong with asserting

$$\forall_{A,B,C,i \in \mathbb{N}} : i > 2 \Rightarrow A^i + B^i \neq C^i$$

except that even very clever evaluation strategies would be unlikely to obtain a definitive answer in a human lifetime.[3]

The following sections survey a number of embedded assertion languages, particularly with respect to the ways in which they address these issues.

3.1 Anna

Anna (ANNotated Ada) [Lv85, LvHKBO87, SRN85, Luc90] is a specification notation for Ada programs. *Anna* is the primary ancestor of many of the more recent executable assertion languages including *ADL* and *APP*, although some of the key features discussed here appeared earlier in an assertion system for the HAL/S language [Tay80, Tay83].

Anna extends the base Ada language[4] with constructs intended to allow a style of programming in which specification and implementation are a continuous process. While our focus in this report is *Anna* as an assertion language for producing test oracles, it grew out of earlier research in formal verification, and was intended to be useful for static verification as well as dynamic testing. Run-time checking was provided for most but not all *Anna* assertions. Program self-checks usable as test oracles are constructed by transforming *Anna* specifications into Ada code for program self-checks. Violation of an asserted property causes the predefined exception ANNA_ERROR to be raised.

Anna annotations are written directly in the source code as "formal comments," i.e., as text that is treated as comments by the Ada compiler but follows syntactic and

[3] The idea for this example (Fermat's last theorem) is due to Richard N. Taylor.

[4] Ada 83, the version of Ada current when the *Anna* project was active

semantic rules that are interpreted by the *Anna* language processor. Formal comments are divided into *annotations* and *virtual Ada text*.

Assertions per se are given in *Anna annotations*, marked as formal comments in which each line begins with `--|`. They are defined using the Ada syntax, extended with quantifiers:

`for all x: T  =>` $P(x)$

means

for all values x of (sub)type T, if $P(x)$ is defined then $P(x)$ is true.[5]

Each annotation: (1) has its own scope, defined by applying Ada scope rules; (2) can use only "visible" entities, that is, either actual or virtual variables[6] that are available within the scope; (3) can be generic:[7] a template for annotating the instantiations of the unit; (4) cannot have side-effects on the actual program.

Anna provides different kinds of annotations for the different Ada constructs. *Object annotations* (Example 3(a)) constrain values of objects within declarative regions. They are equivalent to a set of assertions: one assertion for each line of code that could modify the object. *Type and subtype annotations* (Example 3(b)) constrain a type or subtype: they extend the Ada `range` concept, and like object annotations are equivalent to a set of assertions placed at each point at which an object of a given type can be modified.

Example 3 *Some Anna annotations [Lv85]*

```
M, N : INTEGER := 0;
   --| N <= M;
```

a) The object annotation *requires that* `N` *be always less than or equal to* `M`.

```
subtype EVEN is INTEGER;
   --| where X : EVEN => X mod 2 = 0;
```

b) The type annotation *on type* `EVEN` *requires that each value assigned to a variable of type* `EVEN` *be divisible by 2.*

```
function "/" (NUMERATOR, DENOMINATOR: INTEGER) return INTEGER;
   --| where DENOMINATOR <> 0;
```

c) The subprogram annotation *enforces a precondition requiring that all calls to* `"/"` *have non-zero* `DENOMINATOR`.

```
procedure BINARY_SEARCH(A       : in ARRAY_OF_INTEGER;
                        KEY     : in INTEGER;
                        POSITION: out INTEGER );

   --| where ORDERED(A),
   --|   out (A(POSITION) = KEY),
   --|   raise NOT_FOUND => for all I in A'RANGE => KEY <> A(I);
```

[5] Thus, *Anna* quantifiers extend first order logic quantifiers by being applicable to collections T in which $P(x)$ is undefined for some values.

[6] Virtual variables are variables introduced in virtual Ada text, described below.

[7] A "generic" unit in Ada is similar to a "template" class in C++, but somewhat more flexible in its parameterization.

d) The `out` *keyword introduces a postcondition assertion. The* propagation annotation *ensures that* `NOT_FOUND` *is raised only when* `A` *is ordered and* `KEY` *is not a component of* `A`.

□

Statement annotations specify properties of statements: their scope is determined by the compound statement (block) in which they are declared. *Subprogram annotations* (Example 3(c)) extend the Ada specification part of subprograms and provide a way to clearly state the behavior of a subprogram independently from its body. Such annotations include constraints on formal parameters, results of function calls, and conditions under which exceptions should be propagated. *Exception propagation annotations* (Example 3(d)) specify exceptional behaviors: they annotate exception handlers, `raise` statements, and units that may propagate exceptions. *Context annotations* allow programmers to specify the use of non-local variables within a program unit.

3.1.1 Complex objects

Complex data structures are typically implemented in Ada using packages, much as classes are used in other object-based and object-oriented languages. An operation on a complex object is implemented as a procedure or function of the package (what a C++ or Java programmer would call a "method"). A constraint on a complex data structure is like a structural invariant in verification of an implementation of an abstract data type, in that it should be a precondition and postcondition of each complete operation on the data structure, but need not hold at intermediate points during the implementation of each operation. *Anna* constraints on complex types follow this approach, evaluating the assertions only upon return from each package procedure or function.

In general it is not sufficient to specify a data abstraction by describing the effects of individual operations. Rather, the behavior of a data abstraction is often specified by describing the observable effects of sequences of operations, or by relating the results of different sequences of operations, as in algebraic specifications [GHW85, Gut77]. For the case in which the local data encapsulated in an Ada package is used to represent an instance of an abstract data type, *Anna* provides a way to denote the whole internal state of a package and to denote the state of a package after a sequence of operations.

Example 4 *Anna package axioms [Luc90, pg 157].*

```
package STACK is
  ...
  --| axiom
  --|   for all S: STACK'TYPE; X,Y : ITEM =>
  --|      S[PUSH(X); POP(Y)] = S[POP(Y); PUSH(X)],
  --|      S[PUSH(X); POP(Y)] = S;
```

□

In example 4, `STACK'TYPE` refers to the whole internal state of the package. The first equation states that PUSH and POP operations commute (when both terminate successfully; the axiom does not say anything about what happens if an exception is raised due to stack underflow or overflow), and the second equation states that PUSH and POP are inverses. Ada allows a user-written function to override the equality operation denoted by `=`, so this assertion could compare the states after application

of an appropriate abstraction function, as is usual for describing correctness conditions of abstract data type implementations [Gut77].

The *Anna* axiom notation, together with quantifiers, is expressive enough to state conditions for which a straightforward translation into a run-time check would be unacceptable.

Example 5 *Universal quantification in an Anna axiom [Luc90, pg 159].*

```
for all A, B, N : INTEGER =>
    A mod B = (A+N*B) mod B
```

□

Although the axiom in example 5 is a concise and clear statement of a property of the `mod` operation, one would not like to check the condition by translating it to a triply nested loop over all representable integers.

3.1.2 Virtual Text

Formal reasoning about a program is often facilitated by the addition of variables that are not needed in the actual program computation but are needed as "bookkeeping" for a correctness argument. These have been variously called "ghost variables," "dummy variables," or "auxiliary variables." In *Anna*, ghost variables and computations are introduced in *virtual Ada text* (Example 6), marked with `--:`. The types, functions, and variables introduced in virtual Ada text are visible in other virtual Ada text and in annotations, but not in the Ada program.

Anna allows the bodies of entities introduced in virtual text to be defined using either annotations or virtual text, i.e., Ada code that defines the function body. In the latter case, virtual concepts become an "executable" means for testing and analyzing delivered programs. Virtual text must be legal Ada, with a few additional restrictions. Virtual text treats actual Ada objects as read-only values that it cannot change. Virtual text also cannot hide entities of the underlying Ada program, i.e., the same name cannot refer to two different (virtual and actual) entities.

Example 6 *Anna virtual text [Lv85]*

```
package STACK is
   --: function LENGTH return NATURAL;

       procedure PUSH(X : in ITEM);
          --| where in STACK.LENGTH < MAX,
          --|       out (STACK.LENGHTH = in STACK.LENGTH+1);

       procedure POP(X : out ITEM);
          ...
end STACK;

package body STACK is
   type TABLE is array (POSITIVE RANGE <>) of ITEM;
   SPACE : TABLE(1 .. MAX);
   INDEX : NATURAL range 0 .. MAX := 0;

   --: function LENGTH return NATURAL
```

```
        --| where return INDEX;
   is separate;
      ...
end STACK;
```

Package `STACK` *uses the virtual function* `LENGTH` *to specify the semantics of its procedures; the function is defined through an annotation. Notice also the use of keywords* `in` *and* `out` *in the annotation of procedure* `PUSH` *to define pre- and postconditions, respectively. (The* `is separate` *clause is standard Ada, indicating that the body of the function appears in a different source file.)*

□

3.2 C Assertion Systems

Two assertion systems for C, *APP* and *Nana*, can be viewed as partial re-implementations of *Anna* for the C programming language. The contributions of *APP* are primarily methodological, i.e., in the study of how an assertion system can be effectively used (particularly for software testing), and what features are really needed for effective use, rather than invention of new features. *APP* and *Nana* are also representative of two markedly different implementation approaches for assertion systems. *APP* is implemented as a pre-processor, and features such as value caching are implemented primarily through translation. *Nana*, in contrast, is integrated with a particular program debugger and exploits debugger features to implement value caching.

3.2.1 APP

APP (Annotation Pre-Processor) [Ros92, Ros95], developed in and for the UNIX environment, has the same command interface as *cpp*, the standard pre-processor of UNIX C[8] compilers. It extends `cpp` with the capability of "expanding" the annotations associated with programs. *APP* recognizes assertions written as special comments: they must be enclosed in `/*@ @*/` (Example 7). Comments can be nested in assertions following the C++'s syntax: a comment starts with `//` and ends at the end of the line.

Example 7 *An APP specification of function* `square_root` *([Ros95])*

```
int square_root(x)
int x;
/*@
   assume x >= 0;
   return y where y >= 0;
   return y where y*y <= x && x < (y+1)*(y+1);
@*/
  ...
```

The assertions state that, before executing the function, `x` *must be non-negative; after execution,* `y` *must be non-negative and* `x` *must be between* y^2 *and* $(y+1)^2$*, inclusive.*

□

Assertions are defined using a slightly modified version of C's expression language. *APP* forbids assignments, but adds the operator `in` and iterators. Assignments, and

[8] The examples here are taken from Rosenblum [Ros95] and use the older K&R dialect of C, rather than the newer ANSI C.

assignment-like operators (e.g., `+=` and `-=`), cannot be used to avoid side-effects. Assertions should only evaluate program states and not change them. The operator `in` is used to require that an expression be evaluated in the entry state (before state) of the function that contains the expression.

Assertions can be classified by the point(s) at which they are evaluated. Preconditions are introduced by the keyword `assume`, postconditions by the keyword `promise`, postcondition constraints on returned values by the keyword `return`, and constraints on intermediate states are specified using the `assert` keyword. These correspond to the specifications of assertions one would encounter in a Floyd- or Hoare-style program correctness argument [HK76].

Iterators are used to extend expressions with bounded quantifiers (Example 8). Like C's `for` loops, an iterator has a quantified variable, a condition, and an expression that computes the next value in the collection.

Example 8 *An APP specification of function* `sort` *([Ros95])*

```
int* sort(x, size)
int* x;
int size;

/*@
   assume x && size > 0;
   return S where S &&
      all (int i=0; i < in size-1; i=i+1) S[i] <= S[i+1] &&
      all (int i=0; i < in size; i=i+1)
         some (int j=0; j < in size; j=j+1) x[i] == S[j];
@*/

  ...
```

Before execution (`assume` *clause), the array (*`x`*) must not be null and* `size` *must be positive. The execution (*`return` *clause) should produce an array* `S`*: (1)* `S` *should not be empty; (2) all its elements should be ordered (i.e.,* `S[i]` $\leq$ `S[i+1]`*); (3) each element of* `x` *should correspond to an element of* `S`*, i.e.,* `S` *should be a permutation of* `x`*. Notice that the* `return` *clause uses the operator* `in` *to compute the value of* `size` *before execution.*

□

Evaluating quantifiers as program loops can significantly impact execution time when collections are large or quantifiers are nested. Note that the postcondition assertion for `sort` involves nested quantifiers to check the permutation condition. This check requires a number of comparisons quadratic in the size of the array. Even so, the permutation check as written is sufficient only if the array contains no duplicate elements; the output array is required only to contain at least one representative of each value in the input array. While one might find a more efficient check for the permutation condition, in general it can be difficult to devise assertions that are both clear statements of intent (useful as specifications) and also efficient to evaluate.

3.2.2 Nana

Nana [Mak98] is a library for assertion checking and logging for the GNU C/C++ environment. *Nana* borrows concepts and ideas from other projects (*Anna*, *APP*, and

ADL), existing programming languages (Eiffel), and formal methods (Z and VDM) to deliver an efficient solution to programming with assertions. Quoting the author, *Nana* is "*a nice little library implementing some old ideas in a hopefully useful form.*" The hope is to push these concepts to common good practice.

Assertions can refer to both before and after states. Universal ($\forall$) and existential ($\exists$) quantifiers – together with other macros (e.g., $\exists_1$, $count$, $\sum$, $\prod$) – are supported as defined in the *GNU* library `Q.h` (Example 9). *Nana* also supports C++ iterators as provided by the *Standard Template Library* [MS96].

Example 9 *A Nana specification of function* `qsort` *([Mak98])*

```
void qsort(int v[], int n) {
   I(v != NULL && n >= 0);
   L("qsort(%p, %d)\n", v, n);

    /* the sorting code */

   I(A(int i = 1, i < n, i++, v[i-1] <= v[i]));
}
```

Function `qsort` *sorts the elements in* `v[0..n-1]`. *Before sorting, the first* `I` *clause checks the validity of input parameters and the* `L` *clause logs the current values of* `v` *and* `n` *to a circular buffer. After sorting, the second* `I` *clause verifies that all elements in* `v[1..n-1]` *are sorted. Using more traditional logic notation, the last assertion would be:* $\forall\, i : i \geq 1 \wedge i < n \Rightarrow v[i-1] \leq v[i]$.

□

Assertion checking can be programmed either using simple C code or exploiting the debugging facilities provided by *gdb* (Example 10). Auxiliary variables and value-caching for postconditions can be implemented using *convenience variables*, dynamically typed global variables provided by the *gdb* debugger.

Example 10 *Two Nana specifications of function* `isempty` *[Mak98]*

```
bool isempty(){
   DS($s = s);

   /* code to do the operation */

   DI($s == s);
}
```

a) Using convenience variables, before execution, the value of `s` *is copied in the convenience variable* `$s`. *After execution, the current values of* `s` *and* `$s`, *i.e., the value of* `s` *before execution, are compared.*

```
bool isempty() {
   ID(int olds);
   IS(olds = s);

   /* code to do the operation */

   I(olds == s);
}
```

b) Using pure C, before execution, the value of `s` *is copied in the dummy variable* `olds`. *After execution, the current values of* `s` *and* `olds`, *i.e., the value of* `s` *before execution, are compared.*

□

3.3 Eiffel

Eiffel [Mey97, Mey92] is the most well-known programming language in which assertions are a built-in language feature. In this report we limit our attention to the support Eiffel provides to programming with assertions, touching on other features of Eiffel as an object-oriented programming language only insofar as they interact with assertion features. Eiffel encourages a *design by contract* methodology, interpreting relations among routines (methods) of a system as *contracts* between clients (callers) and suppliers (routines). Assertions provide a way of precisely stating and checking the contracts that govern cooperation among classes in a system.

Eiffel assertions can be used as program documentation, encouraging their use as a primary form of specification. Class interfaces, called *short forms*, can automatically be generated by removing all non-exported features and implementation details from the source code. Short forms contain only public properties and assertions and are a valid means to understand a program without reading the whole code. After defining assertions, actual implementations can be enclosed in `do` clauses or else postponed using keyword `deferred`. In the latter case, Eiffel becomes a (low-level) specification language with which users define the semantics of their operations, omitting implementation details.

Eiffel assertions serve naturally as test oracles, and can be considered to be oracles derived directly from interface specifications, to the extent that the expressiveness of the assertion language is sufficient to capture interface specifications. As with other assertions systems, though, assertions in Eiffel reflect trade-offs between expressiveness and cost. Eiffel assertions (Example 11) must be boolean expressions. To avoid performance problems, Eiffel assertions cannot include sets, sequences, or quantifiers.

Eiffel assertions can refer to classes, single routines, and individual statements. The `invariant` keyword introduces assertions that each class instance must always satisfy, i.e., a condition that should be established by creation of an object instance and maintained by each operation. The `require` keyword introduces preconditions, i.e., requirements that the callers (clients) of a routine must satisfy. Similarly, keyword `ensure` identifies postconditions, the conditions that the routine (provider) guaranties on return. Keywords `check`, `invariant`, and `variant` can be used to state assertions on particular points in execution. An assertion that must hold at a given point in the code is defined in a `check` clause; the condition that must be satisfied as long as a loop is executed, and the condition for loop termination are defined in `invariant` and `variant` clauses, respectively.

Example 11 *An Eiffel class* `ACCOUNT` *([Eif])*

```
class ACCOUNT

feature
   balance: INTEGER;
   ...
   withdraw(sum: INTEGER) is

      require
         sum >= 0
         sum <= balance
      do
```

```
      add(-sum)
   ensure
      balance = old balance - sum
   end -- end WITHDRAW
...
invariant
   balance >= 0
end
```

Class ACCOUNT *has one integer attribute,* balance*, and one routine,* withdraw*. The class invariant imposes that* balance *must always be non-negative for each instance of the class. The precondition of the routine requires that the* sum *to be withdrawn be between* 0 *and* balance*, inclusive. The post condition ensures that the new value of* balance *is its old value less the withdrawn* sum*.*

□

Since Eiffel is an object-oriented language, its assertion sub-language must be reconciled with inheritance, polymorphism, and late binding. The basic problem is to ensure that, wherever an object of subclass B can be substituted for an object of class A, subclass B honors the contract of class A.

The invariant of an Eiffel subclass is the conjunction of its local invariant with all the invariants of its superclasses. Pre and postconditions must always be redefined in a way that ensures subclasses are substitutable for their respective superclasses. If r_A is a routine (method) of class A and r_B its redefinition in B, subclass of A, pre_{r_B} can only be equal to or weaker than pre_{r_A}; dually, $post_{r_B}$ can only be equal to or stronger than $post_{r_A}$. Thus, when at run-time a call to r_A becomes a call to r_B, the precondition of r_B is "weak enough" to be satisfied by any caller that satisfies the precondition of r_A, and the postcondition of r_B is "strong enough"to satisfy any caller that relies on the postcondition to r_A.

Assertions are integrated with the exception handling mechanisms of Eiffel. If a monitored assertion is violated at run-time, it raises an exception that could stop execution or trigger a recovery action. When assertions are used as test oracles, usually the only desirable "recovery" is production of a diagnostic message before halting execution, but the same facility can be used in a production version of the software to establish a stable state after an unanticipated event.

In addition to disallowing quantifiers in assertions to limit the expense of run-time checking, Eiffel provides the programmer control over which assertions will be monitored for each class. The programmer may specify at compile time no-check, preconditions only, postconditions only, pre and postconditions, or everything. As with other assertion systems, this provides the possibility of using assertions as test oracles and debugging aids without incurring the full expense of monitoring in delivered systems.

3.4 Java Assertion Systems

The wide diffusion of Java has motivated several efforts to provide assertion facilities for the Java language. Some aspects of these systems address idiosyncrasies of the language, but many address general problems in adapting assertion systems to modern object-oriented languages and programming styles. They range from simple Java

packages offering an `assert` method to re-implementations of the Eiffel assertion facilities, and from pre-processors to systems that exploit low-level communication with the Java virtual machine to insert assertions on-the-fly during execution. In addition to features and design strategy, these systems also vary greatly in maturity level: some are robust, usable products, while others are research prototypes (some only partially implemented).

Several different systems are summarized in Table 1. The main variations can be summarized as follows:

- Nearly all Java assertion systems are based on pre-processors to transform annotated code into Java source code. Only two are not: *Handshake* uses low-level services to add on-the-fly assertions to code just before being executed by the virtual machine, and the *Java Specification Request (JSR) #41* is a proposed extension to the Java language itself.
- Many systems embed assertions as special-purpose comments, which the pre-processor transforms into Java code. *jContractor*, alone among the pre-processor implementations of assertion processors, interprets specially named methods as pre- and postcondition assertions. A few provide a library package with `assert` methods which can be called like a normal Java method.
- All the listed assertion facilities support the definition of pre- and postcondition checks and invariants, at least indirectly, but only a few also support assertion checks at arbitrary points in control flow (including loop invariants). Systems that provide an `assert` method often do not have special support for pre and post conditions or object invariants, instead requiring the user to place calls to `assert` at the point where the pre- or postcondition should be evaluated.
- It is natural to refer to "previous" values in postconditions, and also to refer to the value returned by a function (which is anonymous in Java), but only a few assertion facilities provide a way to directly refer to them.
- Only a few systems support universal and existential quantifiers directly. In many cases, the programmer must simulate the effect of quantifiers using loops.

Among current Java assertions systems, *iContract* provides most of the important features in their most typical form (as special comments transformed by a pre-processor into executable Java), and it is also among the most widely known and used systems. Rather than describing each assertion system individually, therefore, we present *iContract* as a representative of the whole family of Java assertion facilities.

3.4.1 iContract

iContract provides an Eiffel-like assertion facility for Java. As in *Anna*, assertions are embedded in source code comments, which are transformed by a pre-processor into executable Java code.

Example 12 illustrates the general form of *iContract* pre- and postconditions. The *iContract* keyword `@invariant` specifies class and interface[9] invariants and `@pre`

[9] Hereafter, classes and interfaces are collectively referred to as *types*.

Table 1: Java assertion systems

Approach	invasive	pre-processor	comments	package	pre/post style	check	quantifiers	old value	returned value
iContract ([Kra98])	•	•	•	-	•	-	•	•	•
JaWA/Jass ([jas])	•	•	•	-	•	•	•	•	•
Handshake ([DH98])	-	-	-	-	•	-	-	-	-
jContractor ([KHB98])	•	•	-	-	•	-	-	•	-
JMSAssert ([JMS])	•	•	•	-	•	-	-	•	•
JPP (IDebug) ([KC98])	•	•	•	-	•	-	-	-	•
JML ([Bho00])	•	•	•	-	•	•	•	-	-
corejava Assert ([ass])	•	-	-	•	•	•	-	-	-
JUnit ([jUn])	-	-	-	•	⋆	⋆	-	-	-
JSR ([jsr])	L	-	-	-	⋆	⋆	-	-	-

•: directly supported; ⋆: indirectly supported; L: language extension

The meaning of the columns is as follows:

Invasive: We categorize a system as *invasive* if assertions are embedded in the source program (e.g., as comments or as method calls). Assertions are non-invasive if they are external, not part of the program text.

Pre-processor: The assertion system is implemented as a pre-processor that transforms an extended Java program into a pure Java program.

Comments: Assertions are encoded in special Java comments.

Package: Assertions are provided through a library package or classes with `assert` methods.

Pre- and postcondition, invariant: Support for method pre- and postconditions, and for class invariants. Direct support means that specifications can be placed together as a kind of explicit "contract" for a class or module. Indirect support means that the programmer must place assertions within the control flow of methods, at the point where they should be evaluated.

Check: Simple assertion statements, placed by the programmer in the normal control flow of a method.

Quantifiers: Assertions may quantify universally or existentially over elements of a collection.

Old value: Post-condition assertions can refer to the values of variables before a method call (implying that the assertion system saves a copy of relevant values) as well as current variable values.

Returned value: The result returned by a method can be referenced in a post-condition assertion.

and `@post` specify pre and postconditions for methods. After the keyword, programmers can write any Java expression that returns a boolean. *iContract* extends Java operators with `exists`, `forall`, and `implies`. The first two operators are the existential and universal quantifiers and can iterate over `java.util.Enumeration`, `java.util.Collection` (Java 2), and `java.util.Vector` directly (without having to convert everything to `Enumeration` first). It is also possible to iterate over arrays of object types and primitive types and over ranges of integers and other primitive types. The `implies` operator is shorthand: C `implies` I is transformed into *if C then check I*.

Multiple invariants for the same class, or multiple pre- or postconditions for the same method, are conjoined to form a single aggregate expression. After each expression, programmers can also specify the class that is to be used to construct the exception that is thrown if the assertion does not hold. If no class is defined, class `RunTimeException` is used.

Example 12 *Some iContract annotations ([Kra98])*

```
/**
 * @pre i >= 0         #ArrayIndexOutOfBoundsException
 * @pre i < this.SIZE  #ArrayIndexOutOfBoundsException
 */

String getEntry(int i) throws ArrayIndexOutOfBoundsException {
  // no need to manually check index bounds
}
```

The precondition constrains the value (`i`*) that identifies the positions of available entries in the array.* `i` *must be greater than or equal to* 0*, but it must also be less than the array size (*`this.SIZE`*). If one of the preconditions does not hold, class* `ArrayIndexOutOfBoundsException` *is used to construct the exception.*

```
/** Append an element to the argument
 *
 * @post list.size() == list.size()@pre + 1;
*/

void append(Vector list, Object o);
```

The size of the list after insertion must be one more than the size before insertion (the saved value of which is available using the keyword `@pre`*).*

```
/** Each employee must be in the employment list of all his employers
 *
 * @invariant employees != null
 *    implies
 *       forall Employee e in employees.elements() |
 *          exists Employer c in e.getEmployers() |
 *             c == this
 */
class Employer {
   protected Vector employees; // of Employee
   ...
}
```

The invariant of class `employer` *specifies that each employee must be part of the list of employees of all his employers. The specification uses quantifiers to state that if the list of employees (*`employees`*) is not empty, then for all employees* `e` *in the list there must exist an employer* `c`*, obtained by calling method* `getEmployers` *on* `e`*, such that* `c` *and the "owner" of the list (*`this`*) are the same object.*

□

Class invariants can access class and instance variables as well as methods of their associated classes. Interface invariants have almost the same scope, but they cannot access instance variables. Preconditions can access all properties that are in the scope of their associated method. Besides this, postconditions can refer to a pseudo-variable called `return`, which identifies the result value of the method, and pre-invocation values, that is, the values that the expressions had before invocation: `@pre` appended to any expression e represents the value of e before executing the method. Invariants, pre and postconditions have access to bound variables of quantified expressions. If invariant, pre and postconditions refer to instance variables, the variables must not be private unless the class is final.

iContract manages all four type extension mechanisms (class and interface extension, interface implementation, and inner classes) in the same way. If type T_b extends type T_a, all invariants and conditions defined in T_a apply to T_b as well (Example 13). If T_b defines "local" invariants and conditions, *iContract* merges inherited and local constraints using the following rules: The invariant is the conjunction of the local invariant and all inherited invariants, because subtypes must comply to all restrictions of their supertypes. Postconditions are likewise conjoined, because refined methods must offer at least the functionality of inherited methods. In contrast, the precondition is the disjunction of the local precondition (if any) and all inherited preconditions, because redefined methods must accept at least the input arguments of the inherited method.

Example 13 *Sample propagation of pre and postconditions between interface* `Person` *and class* `Employ` *([Kra98])*

```
interface Person {
   /**
    * @post return > 0
    */

   int getAge();

   /**
    * @pre age > 0
    */

   void setAge(int age);
}

}
class Employ implements Person {

   protected int age_;

   public int getAge() {
      return age_;
   };
```

```
    public void setAge(int age) {
       age_ = age;
    };
}
```

Pre and postconditions defined on the interface `Person` *are implicitly propagated to the implementation class* `Employ`.

□

At run-time, *iContract* requires that the preconditions associated with the chosen constructor hold before creating an object. Both class invariants and constructor postconditions must hold when the object exists. If the constructor fails and an exception is thrown, the class invariants and postconditions do not have to hold. After object creation, *iContract* distinguishes between calls on public, package, and protected methods and on private methods. In the first case, both preconditions and invariants must hold; in the second case, invariants are not checked.

To improve flexibility, *iContract* allows private methods to temporarily violate class invariants. It requires that class invariants hold before exiting the public method [10] that triggered the private ones. In both cases, postconditions are checked before exiting the method. If the execution fails due to a thrown exception, class invariants must still hold, but postconditions are not enforced. *iContract* does not require any constraint on object destruction (i.e., on method `finalize`).

4 Extrinsic Interface Contracts

The assertion languages considered in the previous section provide a way to embed some checkable interface specifications, and possibly other checking, within the program to be tested. In this section we consider systems which provide checkable specifications at a similar level of detail, and which can be used as test oracles in roughly similar circumstances (for unit and subsystem testing, but not for overall system testing) but which keep specifications separate from the implementation. Although it can be considered a trivial variation to keep interface specifications in a separate file or to embed them into source code, the separation is typically coupled with more significant differences. In particular, extrinsic specifications are typically written in notations that are less closely tied to the target programming language, and can even be programming language-independent, and they are more likely to be tied to a particular specification language.

4.1 ADL

ADL (Assertion Definition Language) [SH94] is a language framework designed for testing software components. In contrast to assertion languages embedded in particular programming languages, *ADL* is a meta-notation, i.e., a set of general-purpose concepts that can be rendered into the syntaxes of different programming languages.

[10] In this context, Java's package and protected methods are also considered "public" for the purpose of determining when assertions must be checked.

ADL specifications are not inserted in the program under test, but are placed in separate units. This "non-intrusive" approach makes it possible to associate assertions with pre-compiled code, such as existing libraries or operating systems. Developers must define the bindings between the specifications and the functions in the program. *Assertion checking functions*, that is, the test oracles, are then generated automatically. An assertion checking function is a wrapper around the function under test (f) that, besides calling f, evaluates the associated specification to determine whether f executes correctly.

ADL specifications define post-conditions on their corresponding procedures (functions, methods, etc.). They are partial in the sense that developers need specify only what they want to test; further details can be added to the specification as informal comments. They predicate only on "after" states, that is, the states reached after execution. "Before" states can be referred to using the call-state operator (`@`) that supplies the values of variables at the time the procedure is called (value-caching). Although early versions of *ADL* did not support quantifiers, current implementations provide both universal (`forall`) and existential (`exists`) quantifiers.

ADL was first instantiated as *ADL/C* for the C programming language [Mic93]. Subsequently, *ADL 2* [Pro] extended the original proposal to cover object-oriented notations and interface description languages. Viswanda and Sankar present a preliminary design for *ADL/C++* [VS96], later expanded by Obayashi et al [OKMM98], who describe the new instantiations for C++, CORBA IDL, and Java, and highlight interesting peculiarities of *ADL/Java*. The following sections briefly describe the C and Java instantiations of *ADL*.

4.1.1 ADL/C

ADL/C [Mic93] tailors the *ADL* framework for the C language interfaces. It allows users to describe the behavior of C language interfaces or interfaces readily callable from C, generate documentation, and automatically derive test implementations of the interfaces.

An *ADL/C* specification consists of a set of *modules* (Example 14). Each module can contain *type constituents*, *object constituents*,[11] and *function constituents*. Function constituents contain *semantic descriptions*, which are organized in *bindings* and *assertions*. Bindings define short names to refer to whole expressions. Assertions define boolean expressions that must be true at the end of the execution. Additionally, *auxiliary functions* define concepts that do not belong to the program under test, but are needed for the purpose of specification.

Example 14 *An ADL/C specification of module* `bank` *([SH94])*

```
module bank {

   int errno;
   int NEG_AMT, INS_FUND;

   typedef int acct_no;
```

[11] The term *object* does not refer to object-oriented technology. It is used in the same sense as in C.

```
   int balance(in acct_no acct);

   int withdraw(in acct_no acct, in int amt)
      semantics {
         exception := (return == -1),
         normal := !exception,
         negative_amount := (errno == NEG_AMT),
         insufficient_funds := (errno == INS_FUND),
         @(amt <0) <:> negative_amount,
         @(amt > balance(acct)) <:> insufficient_funds,
         exception --> unchanged(balance(acct)),
         normally (
            balance(acct) == @balance(acct) - amt,
            return == balance(acct)
         )
      }
}
```

Module `bank` *defines a simple set of constituents for a bank account:*

- *Objects* `errno`, `NEG_AMT`, *and* `INS_FUND` *manage error conditions.*
- *Type* `acc_no` *maps account numbers to integers.*
- *The behavior of function* `balance` *is left unspecified. It can be used by other functions, but its implementation is not automatically checked.*
- *Function* `withdraw` *defines four bindings and four assertions. The first two bindings associate expressions to the special names* `normal` *and* `exception`*, and the remaining two bindings define aliases* `negative_amount` *and* `insufficient_funds` *for particular error conditions. The exception operator* $<:>$ *is used to specify that* `negative_amount` *can be true after execution only if* `amt < 0` *is true before execution. The second use of* $<:>$ *similarly associates a precondition with the* `insufficient_funds` *exception. The third assertion uses the predefined function* `unchanged` *to state that if the function fails, i.e.,* `exception` *is true,* `acct` *remains unchanged. The fourth assertion specifies the behavior when* `normal` *is true: The function decrements the balance of* `acct` *by* `amt` *and returns the new account balance.*

All constituent names are directly bound to C elements with the same names.

□

4.1.2 ADL/Java

ADL/Java extends *ADL* technology to cope with object-oriented concepts like inheritance, polymorphism, overloading, and late binding. *ADL/Java* specifications are organized in hierarchies of `adlclasses` (Example 15), with at most one `adlclass` for each Java class. Developers may provide specifications for a subset of the Java classes in a program. This means that: (1) `adlclass` graphs are sub-graphs of the corresponding Java class hierarchies; (2) An `adlclass` can specify methods that are defined in the corresponding Java class, but also methods that are overridden in or simply inherited by the Java class. Redefined methods can be specified using the `super.semantics` feature, in which case the definition of the given method uses the previous definitions of the same method in all inherited class. The approach is recursive: If a method m has already been specified twice in superclasses C_1 and C_2, then

calling a redefinition of m in class C_3, subclass of C_2, would evaluate the assertions of m in C_1, then the assertions of m in C_2, and then the assertions of m in C_3.

Exceptions are defined through Java-like `try`/`catch` clauses. `try` statements enable exception catching while evaluating assertions; `catch` statements define alternate assertion groups to be used to "serve" caught exceptions.

Example 15 *An ADL/Java specification of class* `bank` *([OKMM98])*

```
adlclass bank {
   ...
   BankAcct open_acct(long amt) throws negAmtExc, bnkFullExc {
      semantics
      [abnormal = thrown(negAmtExc, bnkFullExc)] {
         amt < 0 <:> thrown(negAmtExc);
         @bankAux.bank_is_full(this) <:> thrown(bnkFullExc);
         if (normal) {
            return.get_balance() == amt;
            get_accts() == @get_accts() + 1;
            bankAux.is_active(this, return.get_acct_num()) == true;
         }
         if (abnormal) {
            unchanged(get_accts());
         }
      }
   }
}
```

The semantic description of method `open_acct` *defines one binding and four assertions:*

- *The binding relates the special name* `abnormal` *to the occurrence of one of the two throwable exceptions, i.e.,* `abnormal` *is true if one of the two exceptions is thrown.*
- *The first assertion states that if* `amt` *is* < 0*, then the method fails (i.e., throws an exception). But, if the method fails and* `negAmtExc` *was thrown, then* `amt` *must* < 0*.*
- *The second assertion states that if the bank is full, then the method fails. But if it fails and* `bnkFullExc` *was thrown, the bank must be full.*
- *The third assertion states that normal executions require that: (1) the amount of the newly opened account, which is identified by keyword* `return`*, be equal to* `amt`*; (2) the number of accounts opened at the bank be increased by one; (3) the new account be active.*
- *The fourth assertion states that abnormal executions should not change the number of opened accounts.*

□

ADL/Java assertions allow also for inline declarations and auxiliary definitions. Inline declarations are ordinary textual macros. Auxiliary imperative definitions can be added using `prologue` and `epilogue` clauses. These are blocks of pure Java code that do not affect the declarative style of assertions. Prologues and epilogues can be global (belong to the compilation unit) or local (belong to the single class).

4.2 TOG

It is potentially advantageous to use an existing specification notation, rather than inventing a new notation just for the purpose of creating test oracles. On the other hand, deriving oracles from "pure" specification language (discussed below in Section 5) is made more difficult by the generality of notations that were not designed for run-time checking. A middle ground is occupied by adaptations of existing specification styles and notations to the particular task of producing interface contracts from which test oracles can be automatically derived. We consider first an adaptation of SCR-style tabular specifications, and then two approaches to adapting algebraic specifications to test oracle generation.

TOG (Test Oracle Generator) [PP98] generates oracles from relational program documentation in the form of *tabular expressions* [PMI94]. The program (function) under test is specified by means of its signature, the external variables it uses, and its semantics in the form of a *specification relation* between initial and final execution states. All this information is grouped in a table called *program specification* (Example 16). The notation and its semantics is based closely on the tabular notations developed for program specification, and particularly for control systems, but is adapted to specifying the concrete interface of procedures in a particular programming language. Some details of this notation are described below.

Example 16 *A tabular specification of function* `find`[12] *([PP98])*

Program Specification

`void find(int B[N], int x, int* j, bool* present)`		
external variables:		
$D_{find} = \underline{true}$ $C_{find} = \underline{true}$		
$R_{find} =$	$(\exists\, i, bRange(i) \wedge$ `` `B[i] =` `` `x)`	$(\forall\, i, bRange(i) \Rightarrow \neg$ `` (`B[i] =` `` `x))`
`j'` |	`` `B[j'] = x ``	$\underline{true}$
`present' =`	`TRUE`	`FALSE`

$\wedge\ NC($`` `B,` `` `x, B', x')$

Auxiliary Predicate Definitions

$$NC(\mathbf{int}\,`a[\,], \mathbf{int}\,`b, \mathbf{int}\,a'[\,], \mathbf{int}\,b') \doteq (\forall\, i, bRange(i) \Rightarrow\, `a[i] = a'[i] \wedge (`b = b')$$

Inductively Defined Predicates

$$bRange(\mathbf{int}\,i) \doteq \begin{cases} \mathbf{I} = 0 \\ G(i) = i + 1 \\ Q(i) = i < (N - 1) \end{cases}$$

User Definitions

```
#include "defs.h"
#define N 10     /* size of array to search */
```

[12] `` `x `` and `x'` have the usual meanings of the value `x` before and after execution, respectively.

The signature of function `find` *is straightforward:*[13] `B` *is the array to search in,* `x` *is the value to be searched,* `j` *identifies the position of* `x` *in* `B`*, and* `present` *states if* `x` *is in* `B`*. The function does not use any external variables. Its specification relation does not impose any restriction on the input domain D or on the "competence set" C (the portion of the domain for which* `find` *should terminate), so these are represented by the characteristic predicate true.*

*R_{find} is the characteristic predicate of the set of acceptable execution summaries, pairs $\langle x, y \rangle$ where x is the starting state and y is a corresponding stopping state. $`v$ and $v'$ denote the value of a variable $v$ in the initial and final state, respectively. The allowed behavior is broken into two cases, depending on whether an element equal to* `x` *appears in the table, and these two cases are described in two columns of the tabular expression.*

The tabular expression is conjoined with the auxiliary predicate NC, which prohibits changes to other variables. Inductively defined predicates like $bRange$ formally denote the fixed point of a set equation, but can be understood operationally as an iterator, e.g., a C language loop of the form `for( i=` I`;` $Q(i)$`; i=`$G(i)$`)`.

□

Specification relations are limited domain relations (LD-relations) described by their domain D, competence set C, and characteristic predicate R. LD-relations are based on a logic that allows partial functions, but ensures that predicates are total [Par93]. Primitive relations are $false$ if one or more of their argument terms is a function application with argument values outside the function's domain. This ensures that R always has a clearly defined value, either $true$ or $false$, regardless of the values of their arguments. But, this means also that the logics does not include the axiom $X = X$, because $f(x) = f(x)$, only if f is defined at x. The developers of tabular specifications claim that this leads to clearer specifications.

According to the definitions given so far, the execution summary of a terminating program P must be in R, P should terminate for all starting states x in C, but P need not terminate for x in $dom(R) \setminus C$. Moreover, if x does not belong to $dom(R)$, the execution summary should not be in R. To check these statements, *TOG* automatically generates test oracles as four C functions: `initOracle` initializes the oracle, `inRelation` evaluates R on summary executions, `inCompSet` and `inDomain` evaluate C and D, respectively, on starting states. Function `inRelation` is the actual oracle; the last two functions can be used to avoid checking non-terminating executions ($x \notin C$) and executions without acceptable results ($x \notin D$).

Specification relations describe actual program interfaces and data structures, which raises some of the same difficulties as in other assertion languages. Caching of "before" values ($`v$ in specification expressions) poses the usual problems when dealing with pointers or complex data structures, and interpretation of inductive predicates as loops (e.g., evaluating the $bRange$ predicate in the example) can be unreasonably expensive. Problems more specific to the approach of adapting a specification language for test oracle generation arise from the inability to express implementation details, such as "a valid block of memory allocated in the heap." Procedures with formal function parameters are not considered as "programs," because they do not determine a set of possible executions.

[13] The signature in [PP98] is slightly different due to a typographical error.

4.3 Algebraic Specifications

The interface specifications considered so far are in the form of assertions that can be evaluated at a particular point in execution, or precondition/postcondition. While balancing expressive power with efficient run-time evaluation poses a challenge, the idea of using these specifications as test oracles by evaluating them at run-time is at least fairly simple in principle. Not all forms of interface specification can be adapted in this simple manner. In particular, the algebraic approach to specifying the behavior of abstract data types presents a challenge, because the conditions that should be true after some particular operation are not given directly, but rather by equating the results of different sequences of operations.

We next consider two approaches to using deriving test oracles from algebraic abstract data type specifications. In the *DAISTS* approach, axioms equating the different sequences of operations are used directly, and they must be used to derive test cases and the test oracle together (they are not fully general oracles for judging the correctness of arbitrary test executions). In the second, more recent approach, a direct implementation of the algebraic definition of the data type (interpreting equations operationally as rewrite rules) is used as an alternative implementation, whose behavior can be compared to the behavior of a more conventional implementation under test.

4.3.1 DAISTS

DAISTS (Data-Abstraction, Implementation, Specification, and Testing System) [GMH81] is an integrated framework for implementing, specifying and testing abstract data types (ADTs) implemented in the object-based [14] language SIMPL-D, a member of the SIMPL [Bas76] family of languages. The SIMPL-D implementation is augmented with a set of algebraic axioms describing the type, and a set of test cases to validate it.

The implementation is a SIMPL-D `class` declaration (Example 17). The `class` defines a set of variable declarations (i.e., the type representation) and a set of functions (i.e., the body).

Example 17 *An excerpt of a SIMPL-D definition of the data type* `stack` *[GMH81]*

```
define EltType = 'int'
define Undefined = '0'

class Stack = Push, Pop, Top, Empty, NewStack,
              StackEqual, Depth, Limit, assign

define StackSize = '20'

unique EltType array Values(StackSize)
unique int StackTop

Stack func NewStack
  Stack Result
  Result.stackTop := -1
```

[14] We call SIMPL-D object-based rather than object-oriented because, although it does provide encapsulated data structure similar to "classes" in an object-oriented language, it does not provide some of the typical facilities of a modern object-oriented language, particularly inheritance.

```
  return(Result)

Bool func Empty(Stack S)
  return(S.StackTop = -1)

Stack func Push(Stack S, EltType Elt)
  Stack Result
  if S.StackTop + 1 = StackSize
    then
      return(S)
    end
  Result := S
  Result.StackTop := Result.StackTop + 1
  Result.Values(Result.StackTop) := Elt
  return(Result)

Stack func Pop(Stack S)
  Stack Result
    if Empty(S)
      then
        return(NewStack)
      end
  Result := S
  Result.StackTop := Result.StackTop - 1
  return(Result)
...

end class
```

□

A *DAISTS* axiom cannot be used directly to check the correctness of an arbitrary operation (method call). However, an axiom can be used to derive a template of a test case together with an oracle specific to that case. Each axiom of the specification (Example 18) is named and has a list of names and types of the used free variables. *DAISTS* converts axioms into code calling on the implementation: The free variables become parameters, and the operations become function calls.

Example 18 *Some axioms for* `stack` *elements ([GMH81])*

```
Empty1:
  Empty(NewStack) <> True;
```

`Empty` *on a new stack must not return true.*

```
Empty2(Stack S, EltType I):
  Empty(Push(S, I)) = False;
```

`Empty` *on a stack* S, *after a* `Push`, *must return false.*

```
Top1:
  Top(NewStack) = Undefined;
```

`Top` *on a new stack must return undefined.*

```
Top2(Stack S, EltType I):
  Top(Push(S, I)) = if Depth(S) = StackSize
                       then Top(S)
                    else I;
```

`Top` *on a stack* `S`*, after a* `Push`*, must return either* `Top(S)`*, if* `S` *was full, or the pushed element* `I`*, otherwise.*

```
Pop1:
  Pop(NewStack) = NewStack;
```

Function `Pop` *on a new stack must return a new stack.*

```
Pop2(Stack S, EltType I):
  Pop(Push(S, I)) = if Depth(S) = StackSize
                       then Pop(S)
                    else S;
```

`Pop` *on a stack* `S`*, after a* `Push`*, must return either* `Pop(S)`*, if* `S` *was full, or* `S`*, otherwise.*

□

At run-time, a *DAISTS* specification acts as a driver program that executes a set of tests.[15] The driver applies each axiom with (separately provided) test data, and compares the execution of the left and right sides of the axioms. If the two executions do not return values that are considered to be equal (according to a ADT-specific equality function), a diagnostic message is printed indicating that the axiom failed.

DAISTS was a pioneering and well-regarded demonstration that test oracles could be derived from formal specifications even when they are not already in a precondition/postcondition form. However, the approach has two important limitations. First, since the axiomatic specification directly describes implementation entities and operations, the implementations must use a functional style, or more to the point, the approach is not applicable to typical implementations of data structures in procedural and object-oriented languages. Second, the approach requires generation of test cases along with oracles, since each oracle is specific to test cases derived from a particular axiom. It cannot be used to derive general oracles for arbitrary test cases.

4.3.2 Self-checking ADTs

A more recent approach to extracting test oracles from abstract data types, described by Antoy and Hamlet [AH00], treats the algebraic specification essentially as an alternative implementation whose behavior can be compared to the conventional "by-hand" implementation. In contrast to *DAISTS*, this approach does not extract test cases along with the test oracles, but the derived oracles are general in the sense that they can be used with test cases obtained by some other method.

The ADT is specified through a set of operations and a set of axioms (Example 19). The specification is converted into C++ code and becomes the so-called *direct implementation* of the ADT (in contrast to a conventional "by-hand" implementation). Constructors[16] become functions that allocate memory for the representation of the ADT and return a pointer to it; axioms are transformed into functions that represent the instances of ADTs as terms and manipulate these terms according to rewrite rules.

[15] DAISTS also provides some summary and coverage information that is not discussed here, since it is not directly relevant to test oracles; interested readers can find these details in the primary description of *DAISTS* [GMH81].

[16] The term "constructor" as used by Antoy and Hamlet includes what some authors have called "mutators" (functions whose arguments may include the ADT being defined).

The rewriting system ensures confluence and termination to make the rewriting process simpler and more efficient than it would be otherwise, though presumably still less efficient than a conventional implementation of the ADT.

Example 19 *Parts of an algebraic specification for the self-checking ADT* `intset` *([AH00]).*

Constructors:

```
empty(integer, integer)
insert(integer, intset)
```

`empty` *creates an empty set by requiring its size and the upper bound for its elements.* `insert` *inserts a new* `integer` *in the* `intset` *set.*

Axioms:

```
empty(M, R) -> ? :- M < 1 or R < M
```

`empty` *must abort (?) if (*`:-`*) either the size* `M` *is less than one or the upper bound* `R` *is less than the size* `M`*. The last constraint comes from the definition of* `intset`*. Since* `intset` *sets do not allow for negative and repeated elements, they cannot have upper bounds that are less than their sizes.*

```
insert(E, empty(_, R)) -> ? :- E > R
```

`insert` *must abort if the element* `E` *is greater than the upper bound* `R` *of the set created using* `empty`*, no matter of the set's size. Symbol* `_` *identifies anonymous variables.*

```
insert(E, insert(F, S)) -> insert(F, S)
  :- member(E, insert(F, S))
```

Adding an element `E` *to a set does not modify the set if* `E` *already belongs to the set. The axioms for operation* `member` *are omitted here but can be found in [AH00].*

```
insert(E, insert(F, S)) -> ?
  :- not member(E,insert(F,S))
     and cardinality(insert(F,S)) >= maxsize(insert(F,S))
```

`insert` *must abort when trying to add a new element* `E` *if* `E` *does not belong to the set, but the size (*`cardinality`*) of the set is greater than or equal to its maximum size (*`maxsize`*). The meanings of*

```
insert(E,insert(F, S)) -> insert(F,insert(E,S))
  :- not member(E,insert(F, S))
     and cardinality(insert(F, S)) < maxsize(insert(F,S))
     and E > F
```

Inserting an element `E` *in a set* `S`*, to which element* `F` *had been added previously, is the same as inserting* `F` *in the set* `S`*, to which* `E` *had been added before. This is true if* `E` *does not belong to the set* $S+\{F\}$*, the cardinality of the set is less than the maximum size, and* `E` *is greater than* `F`*. The last constraint must hold to ensure that the rewriting system is confluent.*

□

User-defined implementations are called *by-hand implementations*[17]. A direct implementation and a by-hand implementation, together with a *representation mapping*, define the *self-checking implementation* (Example 20). The representation mapping

[17] For example, a by-hand implementation of `intset` can be found in a standard reference on C++ [Str86], Section 5.3.2.

is a key component of the approach: It defines the correspondences between the data structures of the by-hand implementation and the abstractions of the algebraic specification.

At run-time, invocation of a method of the by-hand implementation becomes an invocation of the method with the same name of the self-checking implementation. Both the original code and the corresponding rewrite rules are executed. The representation mapping transforms results on the by-hand implementation into their abstract representations. The self-checking code compares these abstract results with the normal form computed by the rewriting system and treats any discrepancy as an anomaly.

Example 20 *Excerpts of the self-checking implementation of class* `intset` *([AH00])*

Class definition

```
class intset{
  absset abstract;
  absset concr2abstr();

//By-hand implementation
  int cursize, maxsize;
  int *x;
public:
  intset(int m, int n);
  ...
```

The self-checking implementation of class `intset` *is the by-hand implementation with two additional private entities:* `abstract` *identifies the direct-implementation and* `concr2abstr` *is the representation mapping. The by-hand implementation defines two integer attributes to store the current and maximum sizes, a pointer to an integer for the data structure, and a set of public methods. In this excerpt we show only a constructor that requires two integers:* `m`*, the maximum size, and* `n`*, the upper bound.*

Method definition

```
intset::intset(int m, int n) {
  if (m<1 || n<m) error("illegal intset size")
  cursize = 0;
  maxsize = m;
  x = new int[maxsize];

//Additional statements for self-checking
  abstract = empty(m,n);
  verify;
}
```

The self-checking implementation of the constructor, which correspond to constructor `empty` *of the axiomatic specification, is the by-hand implementation with two additional statements. The first statement computes the abstract result by calling function* `empty` *of the direct-implementation. The second statement checks for mutual consistency. Notice that* `verify` *is a predefined macro that compares* `abstract` *with the result from method* `concr2abstr` *and, if needed, prints diagnostic messages.*

Representation mapping

```
absset intset::concr2abstr() {
  absset h = empty(maxsize, MAXINT);
  for (int i = 0; i < cursize; i++)
```

```
    h = ::insert(x[i],h);
  return(h)
}
```

The representation mapping is straightforward. It creates an empty abstract set `h`*, by calling* `empty` *of the direct-implementation, and inserts all concrete elements (*`x[i]`*).* □

While the self-checking approach has been demonstrated with C++ and Java, it addresses the data abstraction facilities of those languages and not other features of object-oriented languages, particularly inheritance or polymorphism. Possibly approaches developed for assertion languages such as *iContract* could be adopted straightforwardly, but this has not been explored. Perhaps a larger question is the extent to which producing an explicit representation mapping creates an additional burden for the programmer, in addition to the burden of creating algebraic specifications.

5 Pure Specification Languages

The test oracles considered in previous sections have been expressed in specification languages designed for run-time checking. They are in this sense not "pure" specifications, i.e., not specifications that one might write first to communicate the intended behavior of a component or system and only afterward use as the source of test oracles.[18] The main additional challenge posed by using a specification language is that effective procedures for evaluating the predicates or carrying out the computations they describe are not generally a concern in the design of these languages. This challenge may be overcome through some combination of restricting the expressions that may be translated to test oracles, transforming some expressions into other, equivalent expressions that are more suitable for run-time evaluation, and requiring the user to hand-code parts of the computation that cannot be derived automatically. The systems for deriving test oracles from Z specifications, discussed next, use all of these tactics.

5.1 Z and Object-Z

Z [Spi89] and its object-oriented variant Object-Z [CDD+89] are model-based specification languages that describe intended behavior using familiar mathematical objects: sets, bags, functions, integers, etc. Being "pure" specification languages, they are free of the constraints of languages designed for efficient computational interpretation. One can describe or quantify over infinite sets as easily as over finite sets, one may negate arbitrary predicates, and one may (and conventionally does) leave many implementation details, including concrete data structures, unspecified. Thus it is not immediately obvious how a Z or Object-Z specification can be interpreted as a test oracle.

[18] One could argue that the tabular SCR-style specification of TOG (Section 4.2) are "pure" specifications in this sense. We have grouped them instead with assertion systems because they refer directly to program entities in the syntax of the implementation language, but there is no clear line between deriving oracles from a "pure" specification language and adapting a specification language to serve as a language for defining test oracles.

The approaches proposed by Mikk ([Mik95]) and McDonald et al. ([MMS97, MS98]) for deriving oracles from Z and Object-Z begin by constraining specifications to an "executable" subset, which can be semi-automatically translated into C or C++. Executability requires that defined types be finite, all predicates be evaluable using a finite number of iterations, and the range of quantified expressions be finite or transformable to a finite one.

Both the Mikk's approach and the approach of McDonald et al. start with a so-called *optimization* phase, analyzing and transforming the original Z specifications into the executable subset. For example, $\exists\, x : \mathbb{N} \bullet x < 10$ (where $\mathbb{N}$ denotes the set of natural numbers) is not in the executable subset, since $\mathbb{N}$ is an infinite set, but using the restriction $x < 10$ the quantifier can be restricted to a finite range. In Mikk's approach, a set of predefined rules (see Example 21) can be complemented with human intervention. Nonetheless it is not always possible to translate specifications into the executable subset, and therefore not all Z or Object-Z specifications can be used as test oracles.

Example 21 *A predefined rewrite rule ([Mik95])*

$$\frac{y \in \{x : M \mid P(x)\}}{P(y) \wedge y \in M} \quad [\ NotOccur(y, P)\ ]$$

Straightforward interpretation of the top formula involves evaluating P for every element of set M. The rewrite rule, which is applicable only if predicate P does not contain y as free variable ($NotOccur(y, P)$), replaces the top formula by a conjunction of two tests involving y.

□

Mikk's predicate compiler also imposes further restrictions: All actual parameters of a generic schema[19] must be constant expressions to determine all its possible instances statically. Each transformable Z schema becomes a C function: Its name becomes the name of the function, the declaration part determines the formal parameters, and the predicate part defines the body of the function. Implementations are built by replacing Z expressions with their values, propositional calculus formulae with truth tables, and quantifications and set constructions with iterations. Partially defined functions[20] are coded using a three-valued logic (undefined $= \perp$) in a strict way ($t_1 = \perp$ is $false$ and $\perp = \perp$ is $false$ as well). Code generation relies on VDM-SL constructs and the *VDM DC* (VDM Domain Compiler, [SH91]) for mapping and converting types. At run time, each function acts as oracle by evaluating its predicates for states passed as parameters and returns a boolean value indicating the outcome of the evaluation.

McDonald et al. adopt a similar approach and propose several alternative transformation of Object-Z specifications into test oracles for container classes ([MMS97, MS98]). After optimizing the specification, they use a special-purpose C++ library, which implements some of the standard types of the Z mathematical toolkit, to transform standard Z types (for example $\mathbb{N}$ becomes `int` and $\mathbb{P}\,\mathbb{Z}$ becomes `Z_set<int>`,

[19] A generic schema is a schema parameterized with respect to a given type, like a C++ template, an Ada generic, or a polymorphic type in ML.

[20] Definition domains of partial functions must be decidable to automatically generate the corresponding code.

where `Z_set<int>` is a template *set* class instantiated using integers). The oracle class and the class under test (CUT) can either be independent classes or related through inheritance. When they are independent, the test driver applies the oracle to the original CUT. When the oracle class extends and inherits from the CUT, it can be invoked automatically (as a "wrapper") to check results produced by the CUT.

Example 22 shows an excerpt of an Object-Z specification that describes the *remove* operation for a class *IntSet* and the C++ code generated as oracle, using inheritance to "wrap" the CUT with the oracle. Each augmented operation provided by the oracle class evaluates the object's state before the real operation, performs it by calling the CUT, and then evaluates the post state. Both pre- and post-state are evaluated in the specification domain by means of a user-supplied translation method `abs()` (abstraction from concrete domain to specification domain) and another method `inv()` that checks the invariant on the abstract state. The abstract states and operations are implemented using the C++ Z mathematical toolkit and are evaluated using simple boolean macros.

Example 22 *Excerpts from the definition of an* IntSet *class ([MS98])*

IntSet

$$\lceil (MAXSIZE, INIT, add, remove, removeAll, isMember, size)$$
$$MAXSIZE : \mathbb{N}$$
$$items : \mathbb{F}\,\mathbb{Z}$$
$$\#items \leq MAXSIZE$$

INIT

$$items = \varnothing$$

...

remove

$$\Delta(items)$$
$$x? : \mathbb{Z}$$
$$x? \notin items \rightarrow removeNotFoundExc$$
$$items' = items \setminus \{x?\}$$

The oracle is a class with a method for each operation, along with a constructor, which corresponds to the INIT *schema, and two particular methods:* `abs` *and* `inv`*. The first method must be coded by hand and defines the translation from concrete to abstract values; the second method codes the class invariant.*

```
class IntSetOracle {
   public:
      int MAXSIZE;
      IntSetOracle(const int maxsize0);
      void remove(const int x);
      ...
   protected:
```

```
      void abs(Z_Set<int>* state_items);
      void inv(Z_Set<int>* state_items) const;
      ...
      void check_remove(const Z_Set<int>* pre_items,
                        const Z_Set<int>* post_items, const int x);
      ...
};

void IntSetOracle::inv(Z_Set<int>* state_items) const {
   CHECKVALBOOLEAN(1, state_items->size() <= MAXSIZE);
}

void InSetOracle::remove(const int x) {
   Z_set<int>* pre_items = new Z_Set<int>();
   abs(pre_items);
   inv(pre_items);
   IntSet::remove(x);
   Z_set<int>* post_items = new Z_Set<int>();
   abs(post_items);
   inv(post_items);
   check_remove(pre_items, post_items, x);
   delete pre_items;
   delete post_items;
}
```

The oracle implementation of the remove *method defines a set of integers, uses it to store the abstract object's state, and checks the invariant against it. Then, it applies the original method, stores the post state in a special-purpose set and checks it against the invariant once more. At the end, it applies pre- and post-state and the removed value* `x` *to the original specification – rendered as a particular method – to investigate their compliance. The* `check_remove` *method simply calls a macro that takes two boolean values and returns an error message if the two values are different. In this case, the comparison is between true (*`1` *in C++) and the fact that the set's size must be less than or equal to the maximum size. Thus, the macro checks whether this holds true.*

□

5.1.1 Test Templates

Stocks and Carrington have described a somewhat different approach for deriving test cases and oracles from Z specifications, considering test program specification and test case specification as a single extended task. Their approach, called the Test Template Framework or *TTF*, is concerned primarily with test case selection, which is outside the scope of this survey. Test oracles can be associated with individual test templates (test case specifications). A hierarchical arrangement of test templates allows the oracle to be either general (near the top of the hierarchy) or specialized to particular test cases.

The approach is illustrated with test oracle templates for the Z specification in Example 23.

Example 23 *Excerpts from the definition of a block-structured symbol table ST ([SC96])*

ST

$st : SYM \pfun VAL$

The table ST is represented as a partial function ($\pfun$) from symbols SYM to values VAL.

$Update0$

ΔST
$s? : SYM$
$v? : VAL$

$st' = st \oplus \{s? \mapsto v?\}$

The definition of operations is divided into basic functionality and error conditions or success messages. $Update0$ defines the basic functionality of $Update$. It adds a new mapping to the table. All operations on the symbol table take the schema ST as argument: Δ represents a state change. $\oplus$ makes $Update0$ add the new mapping $s? \mapsto v?$, if s is not already in the table, or else replace the current value associated with s.

$Success$

$rep! : REPORT$

$rep! = ok$

$Sucess$ simply indicates successful completion of a an operation by setting the rep field to "ok."

$$Update \mathrel{\widehat=} Update0 \land Success$$

The complete definition of $Update$ is then $Update0$ together with $Sucess$.

$$IS_{Up} \mathrel{\widehat=} [st : SYM \pfun VAL;\ s? : SYM;\ v? : VAL]$$
$$OS_{Up} \mathrel{\widehat=} [st' : SYM \pfun VAL;\ rep! : REPORT]$$
$$VIS_{Up} \mathrel{\widehat=} IS_{Up}$$

- *The input set of $Update$ (IS_{Up}) is the before state of st, s, and v.*
- *The output set is the 'after' value of st, that is, st', and rep.*
- *The valid input set is the input set.*

□

An oracle template is a description of the expected outputs given the inputs descrived by the test template: The operation's input is restricted to that defined in the test template and then projected onto the output space of the operation (Example 24). Oracles can define sets of outputs when either test templates are not single-instance templates, or operations are non-deterministic.

Example 24 *A sample test and oracle templates for operation $Update$ ([SC96])*

Using characters for the data type SYM, and natural numbers for VAL, we define a test template $T1$:

$$T1 \mathrel{\widehat=} [s? = \text{'a'} \land v? = 1]$$

The mapping $\{\text{`a'} \mapsto 1\}$ *is a single-instance test template and*

$$oracle_{Up}(T1) == (Update \land T1) \upharpoonright OS_{Up}$$

is the formal definition of the oracle template for operation $Update$ *on test template* $T1$*. The oracle template is the conjunction of the two schemas* $Update$ *and* $T1$*, projected over the output space of* $Update$*. That is,*

$$oracle_{Up}(T1) == [OS_{Up} \mid st' = \{\text{`a'} \mapsto 1\} \land rep! = ok]$$

After applying the single test case represented by $T1$*, the symbol table should contain only the mapping* $\{\text{`a'} \mapsto 1\}$ *and* rep *should be* ok*.*

□

5.2 Temporal Oracles

Specification languages are often specialized to describing particular aspects of program behavior, and abstracting other aspects (which may be described in other specification languages). In particular, specifications based on temporal or interval logics are often used to describe allowable sequences of events, while eliding details of program functionality. These are most familiar from static verification using temporal logic model checkers [CES86, Hol97, CPG00], but temporal specification languages have also been proposed for program specifications. An oracle for a temporal specification language judges the acceptability of a (partially or fully ordered) sequence of events.

Dillon and Yu present an approach for deriving oracles from temporal logics using a tableau method [DR96, DY94]. Oracles for temporal logic formulae are finite-state acceptors that accept sequences of program states, where each state in the sequence is represented by an assignment of boolean values to the propositional variables of the formula. Transitions in the automaton are labeled with (non-temporal) logical formulae that can be evaluated in individual states, so the assignment of truth values in each state determines which transition is taken.[21]

When producing a finite-state automaton, the original specification is associated with the initial state, i.e., it is what must be true from the beginning of execution. The tableau method applies a set of rewrite rules to convert a temporal formula to an equivalent formula. Rewrite rules are applied repeatedly until the original formula is broken down into sum-of-products form. At the conclusion of rewriting, each term of the overall disjunct is a conjunction in which each individual term is either an atomic proposition (possibly negated) which can be checked directly (i.e., evaluated with a particular assignment of propositional truth values from a program state), or else a formula beginning with the "next state" connective; the latter are called "deferred" formulae.

The directly checkable parts of a term becomes the label on a transition leading to a state associated with the deferred part of the term (stripped of the "next-state" prefix). This process is repeated in the new state, and so on repeatedly, to generate all transitions and states of the finite-state acceptor. Because there are only a bounded number

[21] The choice is non-deterministic if the labels of more than one transition may be true in the same program state. Dillon and Yu provide a method for determinising the acceptors, which can theoretically suffer from a combinatorial blowup but which, they report, performs reasonably well in practice.

of formulae that can be generated in this way, eventually the process must generate only states and transitions that have already been generated, and the tableau algorithm terminates. In principle the size of finite state automata can be exponentially larger than the temporal formula from which they are derived, but experience suggests that standard safety and liveness properties usually produce sufficiently compact automata.

A simple example may help the reader obtain an intuition for how a tableau algorithm works; a more complex example that illustrates many more details of the algorithm can be found in Dillon and Yu [DR96]. Suppose the original formula was "eventually p and eventually q," typically written

$$(\Diamond p) \wedge (\Diamond q)$$

$\Diamond p$ ("eventually p") may be rewritten to an equivalent formula,

$$p \vee \bigcirc \Diamond p$$

("p or next-state eventually p") i.e., p is eventually true if it is either true already or if, in the next state, it will eventually be true. The same rewrite rule is applied to the other part of the formula, and after some rearranging one arrives at sum-of-products form:

$$(q \wedge p) \vee (q \wedge \bigcirc \Diamond p) \vee (p \wedge \bigcirc \Diamond q) \vee (\bigcirc \Diamond p \wedge \bigcirc \Diamond q)$$

The four parts of this formula become four transitions. The whole term $(q \wedge p)$ is a formula can be checked directly, so it labels a transition to a state labeled "true," i.e., an accepting state which is reached when the whole temporal formula has been satisfied by a program state or sequence of program states. The term $(q \wedge \bigcirc \Diamond p)$ has a directly checkable part q, which labels a transition to a state associated with the sub-formula $\Diamond p$, i.e., if we observe a program state in which q is true, then the remaining obligation is to find another program state in which p is true.[22]

So far the tableau construction produces a non-deterministic automaton. Note that the two transitions considered so far are not exclusive (q can certainly be true if $p \wedge q$ is true). The last term, moreover, is made up completely of a deferred sub-formulae, and therefore it produces a transition labeled "true." While it would be possible to interpret a non-deterministic automaton, keeping track of all the possible states the automaton may have reached after all possible transitions consistent with each program state, it is also possible to "determinise" the automaton. For the sum-of-products formula above, it is not difficult to see that one can obtain the same ultimate result if the formula is rewritten as

$$(q \wedge p) \vee (\neg p \wedge q \wedge \bigcirc \Diamond p) \vee (p \wedge \neg q \wedge \bigcirc \Diamond q)$$
$$\vee (\neg p \wedge \neg q \wedge \bigcirc \Diamond p \wedge \bigcirc \Diamond q)$$

The rewritten formula produces a deterministic choice among transitions.

[22] For the sake of simplicity, we have glossed over an important detail, viz., determining which automaton states are accepting (final) states. In Dillon's approach, two "next moment" modalities are used, a "weak next" indicating that the sub-formula is satisfied if the execution ends, and a "strong next" which requires further execution. The tableau for $\Diamond p$ requires the "strong next."

5.2.1 Temporal oracles from GIL

Tableau methods have been used for a long time to construct automata useful in temporal logic model checking of finite-state models [Wol85, VW94, GPVW95]. Dillon and Yu observed that the requirements of automata useful as test oracles are somewhat different, since model checking involves reasoning about infinite execution sequences, while test oracles deal only with finite execution sequences. This leads to construction of smaller automata, which can be interpreted as ordinary finite-state acceptors rather than Büchi automata.[23]

The restriction to finite sequences also has consequences for the kind of specification formula that makes sense. Dillon and Yu adopt an interval logic, in which one does not state what must *eventually* happen, but rather state that certain things must happen within some bounded *interval* between two other states [DY94]. The particular interval logic presented by Dillon and Yu uses a graphical syntax that is designed to look like timing diagrams. The GIL graphical syntax is rather unwieldy and is not presented here; the interested reader may find a description and example in the original papers [DKM+94, DY94, DR96].

An execution trace can be considered as a sequence of *events* alternating with an abstract representation of *program states*. For the purposes of checking a program execution against a temporal or interval logic specification, program states are abstracted to assignments of boolean values to the propositional variables that appear in specification formulae, and events include (at least) any program action that changes the value of one or more propositional variable. The input alphabet of the finite-state acceptor is the sequence of program states alone (ignoring the events); the automaton either accepts or rejects a sequence of abstract program states. In practice, an automaton is constructed for the negation of the specification formula, so accepting a sequence of abstract program states indicates that the sequence violates the specified property.

Example 25 *Conversion of an interval logic formula to a test oracle in the form of a finite-state acceptor (adapted from [DY94]).*

$$\Box[[\triangleright\neg\ p\,,\triangleright p\ \|\ \triangleright\triangleright\neg\ p))\Diamond m$$

The above formula requires that propositional variable m must be true in some program state within each interval that begins when propositional variable p changes values from false to true ($\triangleright\neg\ p\,,\triangleright p$), and ending when p becomes false again. This specification is negated to obtain a formula describing a violation of the property:

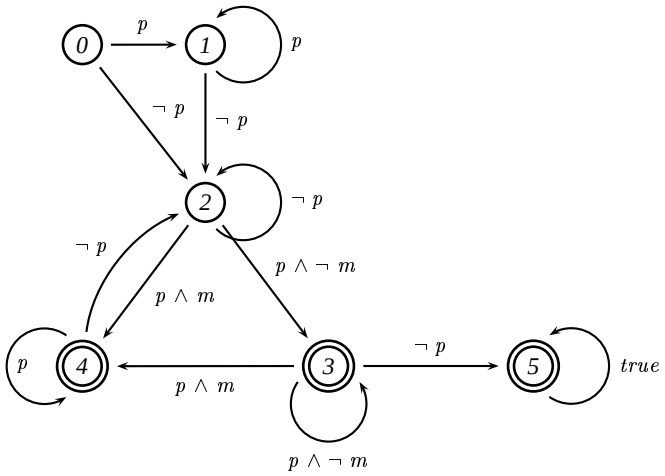
$$\Diamond[\triangleright\triangleright\neg\ p\,,\triangleright\triangleright p\ |\ \triangleright\neg\ p)\Box\neg\ m$$

Negations have been "pushed into" the formula until they are associated with individual propositional variables, which is a precondition for constructing the automaton with a tableau method. Negation changes "eventually" to "always" and vice versa, and also affects the interval constructions. The negated formula can be read as follows: There is some interval ($\Diamond\cdots$) beginning when the truth value of p changes from false to true

[23] A Büchi automaton [Bö60] is like a finite-state acceptor, but instead of accepting a finite sequence if it ends in an accepting state, a Büchi automaton accepts an infinite sequence if it passes through an accepting state an infinite number of times. Infinite sequences can be represented as looping paths in a graph representation, which is how model checkers test Büchi automaton acceptance in finite time.

($\triangleright\triangleright\neg\ p, \triangleright\triangleright p$) and ending when p becomes false again ($\triangleright\neg\ p$), and within that interval m remains always false ($\Box\neg\ m$).

Applying the tableau method to this negated formula to form a finite-state acceptor, and at the same determinising the acceptor, we would obtain:



Each node of the automaton is associated with a set (disjunction) of temporal formulae. Node 0 is associated with the overall formula

$$\Diamond[\triangleright\triangleright\neg\ p, \triangleright\triangleright p\ |\ \triangleright\neg\ p)\Box\neg\ m$$

If p is true in the initial program state, we pass to node 1 which is associated with the disjunction

$$[\triangleright\triangleright\neg\ p, \triangleright\triangleright p\ |\ \triangleright\neg\ p)\Box\neg\ m \vee \Diamond[\triangleright\triangleright\neg\ p, \triangleright\triangleright p\ |\ \triangleright\neg\ p)\Box\neg\ m$$

and so forth. If we encounter a program state in which p is false, and then a state in which p is true and m is false, we enter node 3 of the automaton, in which the end of execution or a state in which p is false will indicate a violation of the temporal specification.

□

5.3 SCR

The temporal oracles described in the previous section are general in the sense that the same oracle can be used for any arbitrary execution, i.e., the oracle is decoupled from test case selection or generation. If the specification is already in the form of a state machine, or can be easily interpreted as a state machine, then it may be useful to derive test cases and corresponding oracles together.

The SCR (Software Cost Reduction, [HKPS78]) method is a requirements specification methodology based on a tabular notation and on several accompanying tools for error detection. Gargantini and Heitmeyer have described an approach for deriving from SCR specification models of external system behavior, oracles paired with test

cases in the form of *test sequences* of inputs and expected outputs. Since the outputs are "computed" from the specification, the SCR model acts as test oracle.[24]

An SCR specification describes both the system and the environment in which the system should operate. The system is represented as an automaton; the environment as a set of *controlled*/*monitored* variables. Changes in the values of controlled variables produce *input events* to which the system reacts by changing state and possibly producing one or more *output events*, that is, changes in controlled quantities. SCR specifications may include also auxiliary variables called *mode classes*, whose values are *modes*, and *terms*. SCR specifications are organized in *event tables* (Example 26) and *condition tables*: the former identify all events the system is sensitive to; the latter define system responses.

Example 26 *The event table that defines the mode class* `Pressure` *[GH99].*

The Safety Injection System (SIS) is a simplified system for safety injection in a nuclear plant. Its inputs are the monitored variables `waterPres`, `block`, *and* `reset`, *and the single output is the controlled variable* `safetyInjection`. *The specification includes also a mode class* `pressure` *whose event table is:*

Old mode	Event	New mode
`tooLow`	`@T(waterPres` $\geq$ `low)`	`permitted`
`permitted`	`@T(waterPres` $\geq$ `permit)`	`high`
	`@T(waterPres` $<$ `low)`	`tooLow`
`high`	`@T(waterPres` $<$ `permit)`	`permitted`

□

The notation `@T(P)` denotes a "transition" into a state in which the predicate `P` is true, e.g., `@T(waterPres` $\geq$ `low)` describes a state in which `waterPres` $\geq$ `low` is not true, but in which it *will* be true in the next state.

The approach of Gargantini and Heitmeyer uses model checking for constructing the test sequences. Given a property $P \Rightarrow Q$ and an SCR specification, test sequences are generated as follows: (a) Instead of $P \Rightarrow Q$, the process starts from the negation of the premise, $\neg P$. (b) The model checker is asked to "verify" $\neg P$. If $\neg P$ were verifiably true, then $P \Rightarrow Q$ would be a vacuous requirement. Since $\neg P$ is not verifiably true, the model checker produces a counter-example. The counter-example is a sequence of events beginning in the initial state and leading to a state in which P is true, which is a suitable state for testing the implication $P \Rightarrow Q$. (c) The trace is used to generate a test sequence by "executing" it on the automaton that represents the system (Example 27.)

Example 27 *An example of test sequence generation [GH99]:*

Suppose the following property:

```
@T(waterPres < low) WHEN block=On ∧ reset=Off
     ⇒ safetyInjection'=Off
```

[24] In Section 4.2 we have already discussed interpreting tabular SCR interface specifications as oracles. We grouped that approach with others that directly described program implementation entities (variables, procedures, etc.), whereas the approach described in this section is based on observable external system behavior. The distinction between "module" and "system" is to some extent arbitrary, and one could certainly group approaches to test oracle generation differently.

The expression states that if `waterPres` *drops below the constant* `low` *when* `block` *is* `On` *and* `reset` *is* `Off`*, then* `safetyInjection` *must be* `Off`.

If the SCR specification of SIS satisfied the property, we could use the property to derive a test sequence. The negation of the premise, stated in temporal logic, is

$$\Box\neg(\bigcirc(waterPres < low) \wedge \neg(waterPres < low \wedge block = On \wedge reset = Off)$$

which is rendered into the input syntax of the SMV model-checker as:

```
AG!(EX(waterPres < low)
 &  !(waterPres < low & block=On & reset=Off)
```

The model-checker produces a trace that generates the following test sequence:

Step	Monitored var. value	Controlled var. value	Mode class value

0	`waterPress=2`
	`block=Off`
	`reset=On`
		`safetyInjection=On`	`pressure=tooLow`

1	`reset=Off`		
2	`waterPress=5`		
3	`waterPress=8`		
4	`waterPress=10`	`safetyInjection=Off`	`pressure=permitted`
5	`block=On`		
6	`waterPress=8`		`pressure=tooLow`

At each step, the table shows only the variable values which change from one step to another. Step 0 corresponds to the initial state and only at step 4 we have a change in the output.

The six-step test sequence can be represented more concisely as:

`<(r,off;-),(w,5;-),(w,8;-),(w,10;s,off),(b,on;-),(w,8;-)>`

where `r`, `w`, *and* `b` *represent the input variables* `reset`, `waterPres`, *and* `block`*;* `s` *represents the output* `safetyInjection`*; and* `-` *indicates no changes.*

□

The approach refers to the properties used to generate test sequences as *trap properties* and proposes a method to automatically select them from SCR tables. Trap properties are generated by covering all possible software behaviors described in the specification. This way the selection process does not require human intervention and the quality of defined test sequences does not depend on who selects the properties. The description of how trap properties are inferred from the tables is out of the scope of this paper; details can be found in [GH99].

5.4 Multi-Language Specifications

Different specification languages are suited to describing different program properties. For example, temporal logics are well suited to describing allowed sequences of events, particularly liveness properties, but they are poorly suited to describing properties of complex data structures. A set of complementary specifications in different formalisms

is often more concise and understandable, and therefore less prone to errors in specification or implementation, than a specification in which all properties have been coerced into one specification paradigm [ZJ96].

One might simply create test oracles from each of the multiple notations used in a program specification. If oracles and test case selection were entirely independent, the cost of independent test oracle generation would be redundancy in monitoring program executions, which might be acceptable. In many cases, though, test oracles and test case selection are coupled, and then one would be limited to applying only one notation-specific set of test oracles with each class of test cases.

Richardson et al. have described a more integrated approach to managing oracles generated from multiple formal specification languages [RLAO92].[25] Oracle information is associated with *test classes*, constraints on test inputs (data or sequences of stimuli) at a sufficiently abstract level that a single concrete test case may satisfy test classes derived from different specifications.

Any oracle derived from a formal specification defines or depends on a relation between entities in the semantic domain of the specification, and observable entities (events, variable values, etc.) from program executions. When there are multiple specification notations, there is one relation for each notation, but those relations may be partly combined. In the approach of Richardson et al., a single execution monitor supports multiple mappings of control and data into the semantic domains of complementary specifications. The monitor essentially gathers state information (or rather, a record of state changes) sufficient to interpret the events and state variables in all of the specifications, through a set of (possibly different) *data mappings*. This information is gathered at points relevant to any of the specifications, as defined by a *control mapping*; some monitored information may be irrelevant to some specifications. Specifications are conjoined by applying each oracle to each applicable control point.

6 Trace Checking

Frequently a partial trace of events is either directly available (at interfaces between a module or system and its environment) or can be obtained through program instrumentation. Such a trace can be checked by an oracle derived from a formal specification of externally observable behavior (e.g., the GIL specifications discussed in Section 5.2), or it may be checked for conformance to a more detailed, operational model of program behavior.

Testing techniques in which sequences of interactions are checked against formal design models[26] have been most thoroughly developed in protocol conformance testing. Characteristics of systems in this domain – concurrency, physical distribution,

[25] The approach is illustrated with Z and a real-time interval logic. Since the contribution of the work is in tactics for combining oracles more than the oracle generation methods for the individual notations, and since in any case the individual oracle generation methods are similar to approaches described elsewhere in this survey, we omit the example.

[26] In the communication protocol domain, these formal design models are conventionally called "specifications." We will follow that convention when discussing communication protocols, but will revert to the term "models" for domains in which they would be a design detail that is not part of the normally observable system behavior.

sensitivity to timing – undermine conventional testing techniques and motivate development of more suitable and special-purpose ones. Moreover, since communication protocols define interfaces over which otherwise opaque components from different organizations must cooperate, they are a suitably complete and precise representation of acceptable behavior.

Oracles based on sequences of interactions can also be applied to distributed component-based software in other domains. The approach is particularly useful for testing systems in which source code is unavailable (but the event trace can be obtained at interfaces between the system under test and its environment). I can be used in integration and system testing, where embedded assertions and specifications of individual interfaces are often inadequate to capture behaviors of interest. For example, an oracle that checks event traces can correlate events in widely separated modules (say, an input and a resulting output) without either a direct interface between the two modules (no "contract" against which interface assertions could be made) or a complete model of how the one event leads eventually to the other.

Trace checking can be applied when no complete specification of acceptable behavior may be available. Rather, one may have only a number of small properties, each of which is believed to be necessary to correct functioning of the system, and which can be checked independently. Satisfying each of these properties does not imply overall correctness, but violation of any one indicates either a faulty program or faulty understanding or formulation of the properties. Some of the properties may be related to overall correctness properties of a program or system, and others may be based on a more detailed operational understanding of *how* the system is intended to work.

6.1 Protocol Conformance Testing

Automatic derivation of test oracles from formal specifications based on state machines was developed earliest and most thoroughly in the domain of communication protocol conformance testing, possibly due to several peculiarities of that domain [vBDZ89, vBP94, FvBK+91]. Enabling factors for development of protocol testing techniques include widespread adoption of a small number of specification formalisms, observability through well-defined interfaces, and functional requirements that do not extend too far beyond what can be modeled using finite-state machines. Usually the same FSM specification is used both as a source of test cases and as a source of information for the test oracle.

Communication protocols are often specified using communicating finite state machines, usually with some extensions that make them not truly finite-state. For example, a state machine that simply receives a message on one port and then sends the same message on another port is not really finite-state unless the set of possible messages is finite. Fortunately, the non-finite-state parts of the specification are often simple enough that an FSM remains a useful model for testing as well as specification. Control systems share some of these characteristics, and the approach described in Section 5.3 is essentially similar, although the state-machine model in that case is derived from SCR specifications.

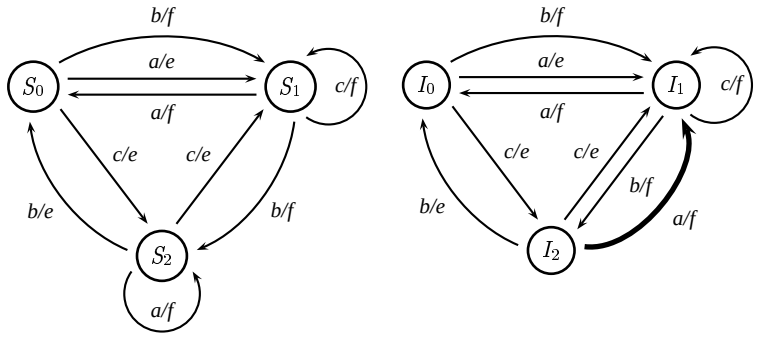
6.1.1 Wp: A representative protocol conformance test method

The *Wp-method* (partial W method) [FvBK+91] is representative of conformance test methods in which test coverage and an oracle are interdependent, and in which (in contrast to most dynamic test techniques) successful execution of a test suite is sufficient to make strong inferences of correctness. Of course, the inference depends on strict limitations on both the formal representation of the protocol and its implementation. They both must be finite, fully specified, and deterministic. They must share the same input alphabet and the number of states in the implementation I must be bound by a known integer m. All states in the two automata must be reachable from the initial states, and the two automata must provide a *reset* operation which returns the protocol to the initial state. If all these hypotheses hold, the method is able to select (based on the specification) a set of test cases that are able to detect (from running the implementation and checking it with an oracle based on the specification) all errors due to both wrong outputs produced by state transitions and transfer errors, that is, different states reached by corresponding transitions in S and I.

Example 28, taken from Fujiwara et al [FvBK+91], illustrates the Wp-method. The specification S and the implementation I meet the requirements of the method, but I is not equivalent to S. Any test suite selected by the Wp-method is therefore guaranteed to expose the discrepancy.

Oracles in the Wp-method are based on sequences of inputs and outputs that uniquely identify states in the specification machine. An *identification set* W_i is determined for each state S_i of specification S; the union of all the W_i is W, a *characterization set* for the automaton. The general strategy is to first use W to characterize each state in the implementation, and afterword use the smaller W_i to check that a transition reaches a particular state S_i.

Example 28 *A simple protocol specification and incorrect implementation (from [FvBK+91])*



(a) Specification *(b) Implementation*

Specification and implementation are clearly different: S has a self-transition on state S_2 labeled a/f (i.e., if a then f), while the same transition in I is from state I_2 to state I_1. For the sake of clarity, reset *transitions are not drawn in the two graphs. S and I have $\{a, b, c\}$ as input alphabet and $\{e, f\}$ as output alphabet. By applying the*

approach, we obtain:

$P = \{\varepsilon, a, b, b.c, b.a, b.b, c, c.a, c.c, c.b\}$
$Q = \{\varepsilon, b, c\}$
$R = \{a, b.c, b.a, b.b, c.a, c.c, c.b\}$
$W_0 = \{a\}$, $W_1 = \{a, b\}$, $W_2 = \{b\}$
$\mathcal{W} = \{\{a\}, \{a, b\}, \{b\}\}$
$W = \{a, b\}$

Phase 1:

Input sequences: $\{a, b, b.a, b.b, c.a, c.b\}$

Output sequences: $\{e, f, f.f, f.f, e.f, e.e\}$

The first phase is not enough to discover the error since the input sequences, applied on the two automata S and I, give the same outputs.

Phase 2:

Input sequences: $\{a.a, a.b, b.c.a, b.c.b, b.a.a, b.b.b, c.a.b, c.c.a, c.c.b, c.b.a\}$

Output sequences: $\{e.f, e.f, f.f.f, f.f.f, f.f.e, f.f.e, e.f.\mathbf{f}, e.e.f, e.e.f, e.e.e\}$

We can identify the faulty behavior of the implementation only by applying the second set of input sequences. The output in bold should have been **e***, this highlights the next state fault of the implementation.*

□

In the example, the output e after input a is enough to distinguish state S_0 from S_1 and S_2, so $W_0 = \{a\}$. Although we might choose $\{c\}$ as a sufficient set of sequences to identify S_1, we (arbitrarily) choose $\{a, b\}$, which (though not minimum among all possible choices) is also minimal, i.e., neither of its subsets $\{a\}$ or $\{b\}$ would be sufficient to distinguish S_1 from S_0 and S_2. Input a with output f is enough to indicate that we are in either state S_1 or S_2, while input b with output f indicates we are in either state S_1 or S_0, so trying both is enough to conclude that we are in state S_1.

To try both inputs from state S_1 it is necessary to reach S_1 once, try a, and later reset to the initial state and reach S_1 again. In general, we will need a set of input sequences sufficient to reach each state, in addition to a set of sequences to distinguish each state from all others. In this case, the *state cover* $Q = \{\epsilon, b, c\}$ is sufficient to reach all the states, and the sequences that reach S_1 and distinguish it from other states are $\{ba, bb\}$. It might seem that we need only $\{cb\}$ to reach and identify S_2, but initially we characterize it with $\{ca, cb\}$ (using all the elements of W). This is because, if we used just $W_0 = \{a\}$ to characterize S_0, and just $W_2 = \{b\}$ to characterize S_2, we might be fooled by an implementation in which a single state I_{02} has transitions a/e and b/e. After having used the entire set W to characterize each state once, we can be sure that the smaller set W_i is enough to identify each state.

The first phase of the Wp method covers each state, but it may not cover all transitions. The second phase of the Wp method covers each of the remaining transitions in S (the transition cover sequences P less the state cover set Q, which was covered in the first phase), using only the sets W_i to check the ending state of each transition.

The heavy constraints on the FSAs imposed in [FvBK+91] have been relaxed in [LPB93] for both partially specified and non deterministic automata. Partially specified automata can be interpreted as complete specifications using either the *don't care* interpretation, which lets the implementation decide the output for undefined transitions, or the *forbidden* interpretation, which considers unspecified transitions as forbidden ones, that is, transitions that cannot be executed. Non deterministic automata further extend the conformance relation to become a *quasi-equivalence* relation: The specification and its implementation must produce the same set of output sequences for every input sequence that can be accepted by the specification. They require also the so-called *complete-testing assumption* (i.e., a finite fairness assumption): by applying a finite number of times a given input sequence to the implementation, it must be possible to exercise all the execution paths of the implementation which are traversed by the input sequence. The number of times an input sequence has to be applied has to be determined using statistical and optimization techniques.

One could argue that the "oracle" part of protocol conformance testing is merely the individual check of an output produced by a transition against that predicted by the specification, and the rest of a method like Wp concerns test coverage. The view taken here is that the "oracle" part is the evidence for concluding that the state reached in the implementation corresponds to the state prescribed by the specification, which may require checking multiple transitions — e.g., the *characterization* of states in the first stage of the Wp method, and the *identification* of states in the second stage. However one chooses to view it, an essential characteristic of protocol conformance testing is the tight coupling between test selection and oracle.

6.2 Oracles for GUIs

Graphical user interfaces (GUIs) are notoriously expensive to test, since usually the behavior of an application must be judged acceptable or not by a human tester. The currently dominant tool-supported approach in commercial practice is capture and replay of execution sequences, which requires a human tester initially but greatly reduces the cost of repeating tests as the software evolves. As regards test oracles, capture/replay with automated comparison to past results is an instance of an oracle based on a set of pre-computed input/output pairs, which is outside the scope of this survey.

An alternative to capture/replay testing of GUIs is automated testing based on a model of GUI operation. When the model can be "executed" through example scenarios, automated testing bears some resemblance to other techniques that drive a system through sequences of stimuli and judge the correctness of the resulting sequences of responses. Primary differences appear in the form of the model which serves as a specification, and in the way that responses are observed. An important enabling condition is that the GUI implementation substrate provide a way of querying a set of properties that describe the state of GUI, rather than requiring interpretation of the actual display.

The Planning Assisted Tester for grapHical user interface Systems (*PATHS* [MPS00]) is a test oracle for graphical interfaces based on an external formal specification of the GUI under test. A set of properties of interest defines the state space to be analyzed; special-purpose operators define how the current state can change. The initial state depends on the chosen test case, as we will see below.

Interfaces can be characterized with either a *complete* or *reduced set* of properties. In the former case, we rely on the graphical toolkit used to implement the interface to identify the properties. For example, if we decided to use Java and its `swing` package, the properties could be all the instance variables associated with objects (classes). In the latter case, designers can define a set of properties that permit them to work at a higher abstraction level. A first set of properties for a notepad-like interface is presented in Example 29.

Actions, that is, state transducers, are used to characterize the evolution of the state of a GUI over time. Since the state space remains implicit, actions are not associated directly with transitions, but they are modeled as *operators*: An action together with its possible parameters, the set of preconditions that must be satisfied to execute it and its effects. For example, if the action were `set-background-color(window, color)`, where the background color is a property for all windows, the precondition could require that the window be active and the current color different from the new one. The effect could require that the color be actually changed (See example 29 for a complete operator.)

Test cases define the initial states and the sequence of operators that must be applied. At each step, the next state is defined by applying the effects associated with the operator to the current state.

Example 29 *Some properties, a possible initial state, and an operator for a simple notepad-like application (from [MPS00])*

`in(file, text)`	*File contains text*
`containsFile(dir, file)`	*Dir contains file*
`currentFont(font, style, size)`	*Defines the current font, style, and size for the document*
`font(text, font, style, size)`	*Text is in font, style, and size*
`onScreen(text)`	*Text is displayed on the screen*
...	

A possible initial state (defined by a particular test case) is defined as follows:

initial:

...
containsfile(samples, f4.doc)
containsfile(private, f1.doc)

currentFont(Times Normal 12pt)

in(f1.doc, **"This"***)*
*font(***"This"***, Times Normal 12pt)*

in(f1.doc, **"is the"***)*
*font(***"is the"***, Times Normal 12pt)*

in(f1.doc, **"text"***)*
*font(***"text"***, Times Normal 12pt)*
...

This excerpt of the initial state identifies two directories, samples and private, and two

files in these directories, f1.doc and f4.doc. Then, it defines the current font used for the document and a few strings that are in the file (text) f1.doc, along with the used font, type, and size.

A simple operator **open** *to open a file of a given directory, is defined as follows:*

Operator Name
$Open(dir : DIRS, file : FILES)$
Precondition
$containsfile(dir, file)$
Effects
$currentFile(file)$
$\neg\ onscreen(obj)\ \forall\ obj \in OBJECTS$
$onScreen(obj)\ \forall\ obj \in OBJECTS \mid in(file, obj)$

The operator states that the opened file becomes the current file. No object (string) is displayed on the screen while opening the file; at the end only the objects in the file are on the screen.

□

The oracle consists of two processes: an *execution monitor* and a *verifier*. The first process is in charge of extracting the current values for all properties of interest. The verifier checks these real values against the expected ones, that is, the ones computed on the external model. The verification can be tailored to different degrees of testing: The verifier can compare only those properties that are expected to change, that is, those properties that are influenced by the effects of the operator. It can check only the properties in the reduced set defined by the designer, or it can check all properties.

7 Log File Analysis

In most software development, the primary specifications and design models are not communicating state machines, and the behavior of interest is not limited to exchange of messages with an environment. The techniques of protocol conformance testing are therefore not completely applicable, but related techniques may still be applicable.

Andrews [And98, AZ00] has described the application of parallel state-machine specifications to check log files produced by application systems. Software developers explicitly include commands to log events of potential interest. Test oracles simulate execution of each individual state machine reacting to only the logged events relevant to that state machine.

The Log File Analysis Language (*LFAL*) [And98] is an explicit description of the states and transitions in a state machine that accepts particular sequences of events.

Example 30 *An LFAL state machine specification describing a property of a graphical user interface [And98].*

```
machine all_popups_get_closed;
  initial state none;
  from none, on open_popup(Name), to open(Name);
  from open(Name), on popup_response(Name, X),
     to exp_close(Name);
  from exp_close(Name), on close_popup(Name),
```

```
   to none;
final state none;
```

□

Example 30 is a *LFAL* specification of a sequencing property of a graphical user interface. It requires that each pop-up window is eventually closed, but only after a user has responded to it. Symbols beginning with upper-case letters are variables which must be consistently bound in the log file, e.g., the example specification would match the open, response, close sequence for pop-up `wp8019` in example 31, but it would fail for pop-up `wp8018` in which the window close event is not preceded by a user response.

Example 31 *A partial log file for an LFAL test oracle. An arbitrary sequence of irrelevant event records may be interleaved with the events shown.*

```
open_popup  wp8018
irrelevent_event foo bar "some text" 24
open-popup  wp8019
popup_response wp8019
exp_close wp8019
menu_open "might be relevant to some other machine"
popup_close wp8018
```

□

In practice, many programmers already create instrumentation that produces traces of "interesting" events for debugging, but they often disable it or even remove it before system testing. Andrews has demonstrated that a collection of simple state-machine specifications can be used as test oracles for log files in a variety of application domains, not limited to those in which state machines are typically used as specifications [AZ00].

Feather [Fea98, FS01] has applied a variant of log file analysis in which the log is loaded into a database and the specified properties are stated as database queries, using the database query engine in lieu of a special-purpose program for checking test oracles. In some cases, existing specifications of application functionality can be created automatically from specifications or design documentation in a domain-specific notation [FS01].

The problem of relating names in a specification to low-level events is less severe in log file analysis, since explicit logging actions are distinct from program functioning and can use the vocabulary of the state-machine specifications. Andrews argues, though, that the content of log files must be carefully designed to capture the relevant semantic events. Essentially the developer is required to design the mapping between events in specified properties and implementation-level events while creating instrumented source code.

8 Discussion

An ideal oracle system would derive test oracles from the same software specification used as the agreement between client and implementer. It would accept "natural" specifications, without imposing constraints that make that specification less useful as documentation and the currency of negotiation. It would nonetheless provide an unerring

pass/fail judgment for any possible program execution, at reasonable cost. Producing any kind of program specification which is comprehensive, precise, and understandable is difficult enough, so it is not surprising that adding effective computability to the set of constraints renders them unsolvable. Thus it is unlikely that there will ever be an ideal system for creating test oracles. Instead, there are a variety of approaches that make different compromises to produce test oracles that, though not ideal, balance the trade-offs to provide useful capabilities.

We have grouped oracle systems based on implementation approaches (e.g., embedded assertions, execution log analyzers) and on the kinds of specifications they accept (e.g., interface specifications, design models, property- and model-based specifications of externally visible behavior.) Imperfect as this classification scheme may be, it does tend to gather together systems that face similar problems, and serves therefore to highlight some recurring strategies and some differences in tactics.

Concrete vs. abstract state and behavior: Some of the oracle systems predicate directly on implementation-level state or observable behavior. These "concrete" oracles include embedded run-time assertions and interface contract assertions, but also log file analysis tools. When oracles are based on more abstract descriptions of program behavior, they must bridge the gap between the concrete entities and specification entities. In one way or another this always involves providing an abstraction mapping from concrete to abstract entities.

Partiality: Oracle systems based on specifications written for other purposes (whether specifications of external program behavior, or module interface specifications) typically try to check those specifications precisely, accepting exactly the behaviors consistent with the specification. When an oracle system uses its own specification notation, distinct from specifications used for other purposes, it is usually "partial" in the sense that only some incorrect behaviors are rejected, and other incorrect behaviors may escape detection.[27] While a complete oracle based on program or module specifications is attractive, partiality has important pragmatic advantages, including low-cost incremental adoption.

Quantification: Specification notations and programming languages make different trade-offs between expressiveness and efficient computability. In a specification notation, there is typically no reason to avoid quantifying over large or infinite sets. Programming languages, on the other hand, either do not provide those constructs or make their costs apparent. A test oracle system, like so-called "executable specification languages," must strike a compromise between expressiveness and efficiency. The range of tactics used by oracle systems, ranging from complete omission of quantifiers to treating quantifiers as looping constructs to attempts to rewrite specifications to eliminate them, indicates that there is no clear optimum balance nor any fully satisfactory approach to accommodating quantifiers.

[27] Weyuker calls these "pseudo-oracles," and notes that it is sometimes much easier to distinguish plausible from implausible results than to precisely distinguish correct from incorrect results [Wey82].

Oracles and test case selection: In an ideal oracle system, oracles would be orthogonal to test case selection. In practical systems, it is sometimes more practical to determine acceptable behaviors for limited classes of test cases. In particular, one must often trade expressiveness of specifications against generality. Often, but not always, the trade-off is partly determined by the nature of the specifications. Model-oriented specifications and design models, which lend themselves to simulated execution, are often used to derive test classes and test-class-specific test oracles together. Property-oriented specifications are more often used to derive test oracles that are independent of test cases, although even a property-oriented specification may sometimes be symbolically evaluated to obtain a simpler or more efficient test oracle for a limited set of test cases.

These same trade-offs are to some extent an indication of areas where one may expect future research progress, tied on the one hand to research in software testing and on the other to research in specification languages and methodologies. In the interim, despite the absence now or in the foreseeable future of an ideal system for creating test oracles, the state of the art and practice is already well enough developed that there can be little excuse for relying exclusively on the most expensive and least dependable of test oracles, the human eye.

References

[AH00] Sergio Antoy and Richard G. Hamlet. Automatically Checking an Implementation against Its Formal Specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[And98] James H. Andrews. Testing using log file analysis: Tools, methods, and issues. In *Proceedings of the 13th IEEE International Conference on Advanced Software Engineering*, pages 157–166, Honolulu, Hawaii, October 1998.

[ass] Package corejava. `http://www.hio.hen.nl/java/corejava/`.

[AZ00] James H. Andrews and Yingjun Zhang. Broad-spectrum studies of log file analysis. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 105–114, Limerick, Ireland, June 2000. ACM Press.

[B$\ddot{\text{o}}$60] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. 1960 Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[Bas76] Victor R. Basili. The Design and Implementation of a Family of Application-Oriented Languages. In *Proceedings of the 5th Texas Conference on Computing Systems*, pages 6–12, October 1976.

[Bho00] A. Bhorkar. A Run-time Assertion Checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 2000.

[CDD$^+$89] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In *Formal Description Techniques (FORTE '89)*, pages 281–296. North-Holland Publishing Co., December 1989.

[CES86] Edmund Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Cha82] D. Chapman. A Program Testing Assistant. *Communications of the ACM*, September 1982.

[CPG00] Edmund Clarke, Doron Peled, and Orna Grumberg. *Model Checking*. MIT Press, 2000.

[DG85] J.D. Day and J.D. Gannon. A test oracle based on formal specifications. In *Proc. SoftFair, A Second Conf. on Software Development Tools, Techniques, and Alternatives*, pages 126–130, San Francisco, Dec 1985. ACM Press.

[DH98] A. Duncan and U. Hlzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, 1998.

[DKM+94] L.K. Dillon, G. Kutty, L.E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[DR96] L.K. Dillon and Y.S. Ramakrishna. Generating Oracles from Your Favorite Temporal Logic Specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21(6) of *ACM Software Engineering Notes*, pages 106–117. ACM Press, October 1996.

[DY94] L.K. Dillon and Q. Yu. Oracles for Checking Temporal Properties of Concurrent Systems. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 140–153, December 1994.

[Eif] Eiffel web page. `http://www.eiffel.com`.

[Fea98] Martin S. Feather. Rapid application of lightweight formal methods for consistency analysis. *IEEE Transactions on Software Engineering*, 24(11):949–959, November 1998.

[FS01] Martin S. Feather and Ben Smith. Automatic generation of test oracles — from pilot studies to application. *Automated Software Engineering Journal*, 8(1):31–62, 2001.

[FvBK+91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[GH99] Angelo Gargantini and Connie Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, volume 24.6 of *Software Engineering Notes (SEN)*, pages 146–162. ACM Press, September 6–10 1999.

[GHW85] John V. Guttag, James J. Horning, and Jeanette M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 8(3):24–36, September 1985.

[GMH81] John D. Gannon, Paul McMullin, and Richard G. Hamlet. Data-Abstraction Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Gut77] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[Ham77] Richard G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[HK76] Sidney L. Hantler and John C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[HKPS78] K. L. Heninger, J. Kallander, D. L. Parnas, and J. E. Shore. Software Requirements for the A-7E Aircraft. NRL Memorandum Report 3876, United States Naval Research Laboratory, November 1978.

[Hol97] Gerard Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[jas] The Jass Page. `http://semantik.informatik.uni-oldenburg.de/~jass`.

[JMS] Using JMSAssert to Design by Contract. `http://www.mmsindia.com/JMSAssert.html`.

[jsr] Jsr41. `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_041_asrt.html`.

[jUn] Junit. `http://www.junit.org`.

[KC98] J.R. Kiniry and E. Cheong. JPP: A Java Pre-processor. Technical Report CS-TR-98-15, California Institute of Technology, 1998.

[KHB98] M. Karaorman, U. Hlzle, and J. Bruno. jContractor: Reflective Java Library to Support Design-by-Contract. Technical Report TRCS98-31, University of California, Santa Barbara, 1998.

[Kra98] R. Kramer. iContract – The Java Design by Contract Tool. In *Proceedings of TOOLS26: Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE Computer Society, 1998.

[LPB93] G. Luo, A. Petrenko, and G. V. Bochmann. Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines. Technical Report IRO-864, Dept. d'Informatique et de Recherche Oprationnelle, Universit de Montral, 1993.

[Luc90] David Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*. Springer-Verlag, 1990.

[Lv85] D.C. Luckham and F.W. von Henke. Overview of Anna, a Specification Language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[LvHKBO87] D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, and O. Owe. *Anna - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Mak98] P.J. Maker. GNU Nana – User's Guide (version 2.4). Technical report, School of Information Technology – Northern Territory University, July 1998.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, 1997.

[Mic93] Sun Microsystems. ADL language reference manual, 1993.

[Mik95] E. Mikk. Compilation of Z Specifications into C for Automatic Test Result Evaluation. In *Proceedings of the 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1995.

[MMS97] J. McDonald, L. Murray, and P. Strooper. Translating Object-Z specifications to object-oriented test oracles. In *Proceedings: 4th Asia-Pacific Software Engineering and International Computer Science Conference*, pages 414–426. IEEE Computer Society Press, 1997.

[MPS00] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of *ACM Software Engineering Notes*, pages 30–39. ACM Press, November 2000.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[MS98] J. McDonald and P. Strooper. Translating Object-Z specifications to passive test oracles. In *Proceedings of the 2nd International Conference on Formal Engineering Methods (ICFEM98)*, pages 165–174. IEEE Computer Society Press, 1998.

[OKMM98] M. Obayashi, H. Kubota, S.P. McCarron, and L. Mallet. The Assertion Based Testing Tool for OOP: ADL2. `http://adl.xopen.org/exgr/icse/icse98.htm`, May 1998.

[Pan78] D.J. Panzl. Automatic Software Test Drivers. *Computer*, 11(4):44–50, April 1978.

[Par93] David L. Parnas. Predicate Logic for Software Engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[PMI94] D.L. Parnas, J. Madey, and M. Iglewski. Precise Documentation of Well-Structured Programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

[PP98] Dennis K. Peters and David L. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[Pro] ADL Project. Future plans: Adl 2. `http://adl.xopen.org/future/`.

[RLAO92] Debra J. Richardson, Stephanie Leif-Aha, and T. Owen OMalley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[Ros92] David S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.

[Ros95] David S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[SC96] Paul Stocks and David Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

[SH91] U. Schmidt and H. Horcher. The VDM Domain Compiler: A VDM Class Library Generator. In *Proceedings of the 4th International Symposium of VDM Europe*, volume 551 of *Lecture Notes in Computer Science*, pages 675–687. Springer-Verlag, 1991.

[SH94] S. Sankar and R. Hayes. ADL: An Interface Definition Language for Specifying and Testing Software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994.

[Spi89] John M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[SRN85] S. Sankar, D. Rosenblum, and R. Neff. An Implementation of Anna. In *Proceedings of the Ada International Conference on Ada in Use*, pages 285–296. ACM, Cambridge University Press, May 1985.

[Str86] Bjorne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tay80] Richard N. Taylor. Assertions in programming languages. *ACM SIGPLAN Notices*, 15(1):105–114, 1980.

[Tay83] Richard N. Taylor. An integrated verification and testing environment. *Software — Practice and Experience*, 13:697–713, 1983.

[vBDZ89] G. v. Bochman, R. Dssouli, and J.R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11):1347–1356, November 1989.

[vBP94] G. v. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. Technical Report IRO-923, Dept. d'Informatique et de Recherche Oprationnelle, Universit de Montral, 1994.

[VS96] S.R. Viswanadha and S. Sankar. Preliminary Design of ADL/C++ — A Specification Langauge for C++. In *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems (COOTS), Toronto, Canada*, June 1996.

[VW94] Moshe Vardy and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *Compuer Journal*, 25:465–470, 1982.

[Wol85] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 110-111:119–136, 1985.

[ZJ96] Pamela Zave and Michael Jackson. Where Do Operations Come From? A Multiparadigm Specification Technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, 1996.