An interface language between specifications and testing

George Fink Michael Helmke Matt Bishop Karl Levitt

August 16, 1995

1 Introduction

Specifications describe models that programs are supposed to implement. Testing is used to verify if a program's implementation is correct. Specification languages such as Z [Dil90] don't provide ways to relate their model to a program state. Alternatively, the program could be developed independent of the specification, so there is no natural correspondence between specification and code.

In this paper, we present a new specification language **TASPEC** which can serve as an intermediary between a Z specification and the testing process. **TASPEC** can express close correspondences between code and abstract semantics. Large portions of Z are shown to be semi-automatically translatable into **TASPEC**. Test oracles and other testing artifacts can be derived from **TASPEC** specifications. As an example, a recently discovered flaw [Guh95] in an implementation of the TCP protocol is uncovered using this technique.

TASPEC has primitive constructs that enables it to be easily translated into slicing criteria and execution monitors. **TASPEC** includes the basic logical and temporal operators, together with location specifiers that allow events to be associated with code features. This provides the primitive data for analyzing higher-level semantic features of the program.

TASPEC is a part of a larger methodology, property-based testing. A property is a specification that is independent of program structure. For example, authentication is just one aspect of a login program; authentication is just a property related to the program. Property-based testing is a method for validating a program with respect to a property, using **TASPEC** specifications of generic properties or generic flaws as a basis for testing programs. Property-based testing is designed as a way to reconcile code with property and flaw descriptions. Property-based testing consists of **TASPEC**, static slicing, path coverage criteria, and execution monitoring. The Tester's Assistant is a prototype implementation for **TASPEC** and property-based testing.

2 Background

The goals behind using specifications in testing are establishing greater formalism for test results and increasing automatability and re usability of test artifacts. Prior methods for utilizing specifications in testing fall into three loose categories: specifications to generate test data, specifications to create test oracles (verifying the correctness of an execution), and specifications refined into code (and therefore having direct specification-code relationships that can be measured).

Specification languages such as Z [Dil90] and VDM [AI91] can be used to fully specify a system at a more abstract level than source code. The data structures and operations can gradually be made more concrete through refinements to the specification, until at some point the boundary between specification and source code is crossed. Presumably, the more abstract specifications better reflect the desired abstract functionality (though they are less specific), so via the correspondences extracted from the refinement process, concrete execution states can be compared to abstract states. This comparison can serve as a test oracle. Such a specification could also be used to generate tests; however this is little different from generating tests from the source code due to the shared derivation of code and specification.

Test oracles can also be automatically generated from other specification-based approaches such as Larch [GH93] and TAOS [RAO92] [Ric94]. Function and procedure behavior is specified as in the refinement methods, but in a separate process from the actual coding. The specifications can then serve as independent test oracles without being influenced by implementation bias. Links between specification objects and implementation object need to be provided, however, so that the respective states can be compared. This linkage can be done easily if the unit of specification is the behavior of individual functions (or modules) in the implementation. Formal parameters can be linked with actual parameters, and so on.

Test data can also be generated from specification artifacts. ADL [San89] [SH94] and TAOS have test description languages whereby test data can be categorized. Once categorization is made, generating exhaustive test data with respect to the structure of the specification is

possible [CRS]. Prior to this work, similar techniques had been used with VDM [DF93]. Related work has also been done to generate test data from the structure of code such as in [GG75] and [How75].

With **TASPEC** and property-based testing, test oracles can be generated independently of descriptions of specific modules or functions. With the emphasis on *properties* and not on full specification, test oracles can be made to handle a wider class of behavior than that rigidly defined by functional specifications. Translations between other specification languages and **TASPEC** can provide additional flexibility to the specification and testing phases of development.

3 Property-based Testing

Property-based testing [Fin95] uses specifications of generic properties in the **TASPEC** language to produce structural tests of specific programs. Through location specifications, properties are associated with code. The code thus designated, and all code related to the designated code, is subject to the testing process. The marked code is monitored for correctness during execution and for test coverage completeness.

Properties are defined independently of a specific program, and so can be grouped together in libraries of properties. These libraries can be reused and also analyzed by independent means to assess their completeness[1].

Structural testing is used to assure that the actual source code of the program adheres to the property specification. For this goal, it is important to know in what ways the program can interact with the property, and to rigorously test these ways. Property-based testing focuses on the generation of test cases from specifications only indirectly; generation of test cases is driven by gaps in code coverage (which is determined in turn by property specifications). Therefore, property-based testing is complementary to other specification-based testing techniques.

Program slicing reduces the amount of source code to be analyzed. This reduction is of immediate value to a human tester inspecting the code manually. Further benefit is gained by applying automatic analysis tools to the slice rather than to the whole program. To calculate a slice, very detailed global dependencies need to be derived; this dependence information is applied to the other analysis tools as well.

Coverage metrics that are both practical and relevant to satisfying validation requirements are used in property-based testing. Given a formula of interest, the optimal metric requires all possible results for that formula; for most formulas this requires a very large (or computationally infeasible) number of data values. Metrics based upon program paths within the slice approximate this optimum, given the testing framework.

Automatic high-level execution monitors are derived from property specifications in **TASPEC**. Primitive events are produced via location specifiers; higher-level events are raised by the execution monitor as dictated by the property specifications. Checking the adherence of a program execution to a complex property specification therefore proceeds automatically.

The Tester's Assistant is a prototype implementation of property-based testing for analysis of C programs; its development is ongoing. Currently operational are static analysis, slicing, and code instrumentation modules. Code can be instrumented both for the purposes of obtaining path coverage information, and also for capturing abstract behavior described in **TASPEC**. The Tester's Assistant(and **TASPEC**) has been applied to UNIX security properties and network server code [FL94].

4 The Z Specification Language

Z is a formal specification language developed at Oxford University that allows software requirements to be specified using a precise mathematical notation. One of the first industry successes using Z was in specifying portions of IBM's CICS system [Kin92]. Since then, Z has gained popularity as a formal tool in both industry and academia world-wide.

In this paper the following terms are used: **Declaration Part** refers to the upper half of a schema where data declarations and schema inclusions are located. **Predicate Part** refers to the lower half of a schema where invariants, preconditions, and definitions are located. **State Schemas** define global data in a system. These schemas usually contain invariants to constrain the data. **Operational Schemas** specify an operation. Such a schema will include one or more State Schemas and declare Input and Output parameters. **Schema Names** come in several varieties. *Undashed* and *dashed* names let the same name indicate the before and after state of an operation. Input parameter names end with a ? and output parameter names end with a !. The terms *pre-names* and *post-names* refer to the set of *undashed* and *input* names and *dashed* and *output* names respectively.

Additional information about Z may be found in [Spi92], [Wor92],[Dil90].

5 TASPEC

TASPEC was created as a specification interface to property-based testing and the Tester's Assistant.

[1] Through a previous iteration of property-based testing, perhaps.

TASPEC needed to be able to describe generic flaws and properties but also needed to have a simple correspondence with code in order for test oracles and property-based coverage criteria to be easily derived. Event-based semantics were chosen for their conceptual simplicity. Event processing was augmented with first-order predicate logic to provide expressive power.

Z and VDM do not provide the correspondence between source code and logic. The non-executable nature of Z makes it difficult to convert into non-trivial test oracles directly. The **TASPEC** language was designed specifically to meet the needs of property-based testing. Furthermore, more sophisticated languages such as Z and VDM can be translated into **TASPEC**, thereby combining the well-known utility of the former languages with the testing machinery of the latter.

5.1 TASPEC language definition

A specification in **TASPEC** describes an abstract state. The abstract state is correlated with program locations. During the program's execution, the abstract state specification is converted into a concrete stream of state events. This state stream is checked against the abstract state for errors.

A concrete state element in **TASPEC** is a user-defined term, e.g., $\mathtt{arrayref}(\mathtt{a}, 10)$. If this element gets put into the state stream, that signifies that during the execution of the program, element number 10 of the array $\mathtt{a}$ was accessed.

TASPEC also can specify invariants that the program must satisfy. The invariants are expressed by linking state events together with standard logical connectives. These invariants together form the abstract state that is compared with the concrete state in order to check for errors.

Multiple assertions with a common identifier can appear in the state at the same time. Most often, for each instance there are different argument values. Consider, however, an example in which an abstract finite set is being represented in **TASPEC**. The set is enumerated by a state element named set, i.e., if $\mathtt{a}$ and $\mathtt{b}$ are to be in the set, the concrete state would hold the two elements $\mathtt{set}(\mathtt{a})$ and $\mathtt{set}(\mathtt{b})$. A very useful operation for this sort of abstract state is the number of different set elements there are in the state. A built-in predicate sizeof returns this number. In the above example, $\mathtt{sizeof}(\mathtt{set})$ would be replaced with the value 2.

TASPEC's semantic model is loosely based upon that of Prolog [CM84]. The state consists of a collection of facts that are tied together by various predicates. The language for expressing invariants is closely related to constructs in Prolog, because the system was designed with Prolog in mind as the engine for monitoring execution.

There are three other components in **TASPEC**: compounding operators, location specifiers, and event processing. This section will explain each of these components in more detail.

5.1.1 Compounding operators

- Arithmetic operations $(+, -, *, /)$
- Relational operators (=,<,>,<>)
- Logic operations $(\cup, \cap, \neg, \Rightarrow)$
- Temporal operations (**before**, **until**, **eventually**)

Arithmetic and relational operators are used to place constraints on data values. For example, consider this fragment from the specification of array bounds (See Section 5.2 for the complete specification):

$$\mathtt{arrayref}(\mathtt{a}, \mathtt{i}) \cap \mathtt{array}(\mathtt{a}, \mathtt{l}, \mathtt{u}) \Rightarrow \mathtt{i} \leq \mathtt{u} \cap \mathtt{i} \geq \mathtt{l}.$$

The values $\mathtt{i}$, $\mathtt{l}$, and $\mathtt{u}$ are bound by their presence in the state elements array and arrayref, which are tied to array declarations and usages respectively.

Logic and temporal operations are used to group simple facts into more complex terms. The logical operators allow any first-order predicate to be expressed in **TASPEC**. The temporal operators allow certain relationships of predicates in time to be expressed easily.

5.1.2 Location specifiers

Location specifiers are essential in tying **TASPEC** specifications to actual source code. Ideally, it should be possible to have location specifiers linked to every point in the program where the abstract state changes. For the purposes of the current implementation, locations are limited to function calls, variable usages, variable declarations, and variable assignments. With this limitation, there is a direct mapping between locations and nodes in the data-flow graph of the program.

A conditional expression can be attached to a location. At run-time, the conditional expression is evaluated when the location is reached, to test to see if the specification should match. For example,

location malloc() **returns** ptr **if** $\mathtt{ptr} \neq 0$

matches all instances of the `malloc` library call that return with a non-zero value.

Function, variable, and assignment locations can be parameterized by copying data values from locations in the code. If a parameter appears in the location portion of a specification, its value is stored and is propagated to the action portion of the specification. For function locations, the arguments to the function as well as the result value can be used as parameters (See Figure 1). For variable locations, array indices can be used.

location $\quad calloc(n, s)$ **returns** $addr$
$\{$**assert** $object(addr, 0, n-1);\ \}$

Figure 1: A result parameter is used in an assertion on the right-hand side of a specification

5.1.3 Event processing

Event processing controls how the concrete state is constructed, and also describes the invariants against which the concrete state should be checked. The event processing grammar will be presented in two parts. First is a description of the mechanisms for altering the concrete state. Then we describe invariants and how the state altering mechanisms are activated.

The state is changed by either asserting a concrete state element (adding it to the state) or retracting (removing) it from the state. The state element consists of an identifier that acts as a label and a list of parameters that make the element more specific. There is an optional iterator, which can be used in setting up larger state constructs. The iterator is of special use when initializing the concrete state with information about an array or similar structure. For example,

assert$(i = 0, n) array_contents(array, i, 0)$

could be used to initialize each element in an array's representation in the concrete state to 0. In this example n and $array$ are variables that are bound in the context of the **TASPEC** fragment.

The fundamental unit of specification is an event. An event can either be a change of state (assert or retract) or be a predicate. A predicate can have one of two possible meanings, depending on whether or not an event list is attached to it. If no event list is attached, the predicate is an invariant. If the program, while executing, produces a concrete state in violation of any invariant, then an error message is printed with the description of the flawed execution.

If an event list is attached, then the predicate is treated in the same way as a location. The predicate/location provides variable bindings; any attached events are processed in that context and only when the predicate or location matches the execution of the program. The grammar is recursive in this extent; any event description can be predicate/location dependent, including a predicate which in turn has nested events.

Consider the following example:

location $\ldots \{array(a, l, u)\{$**assert** $array(b, l, u)\}\}$

The second $array$ element is high level in that the correspondence between the element and a location is not direct; it only raised when the additional event formula is satisfied.

location decl $a[10]\{$**assert** $array(a, 0, 9);\ \}$
location variable $a[i]\{$**assert** $arrayref(a, i);\ \}$
location assign a **result** b
$\{array(a, l, u)\{$**assert** $array(b, l, u)\}$
$arrayref(a, i) \cap array(a, l, u) \Rightarrow i \leq u \cap i \geq l$

Figure 2: Specification of array bounds checking.

5.2 Example TASPEC specification

The complete specification for array bounds checking for statically declared arrays is in Figure 2. A variant of the variable location is used in this case, which matches the declaration of a variable, and not the use of a variable (the use being the default for variable location specifications.) All array declarations and array references cause assertions to be made. An invariant is added that constrains the value of the array index. In some instances, this check can be made at compile time (i.e. analysis/slicing time). Usually, however, the exact equation that computes the index will be too complex to analyze statically and so establishing the correctness of array reference must be done through repeated test runs, using property-based testing. In this case, the constraint becomes a run-time check for validity of a test run.

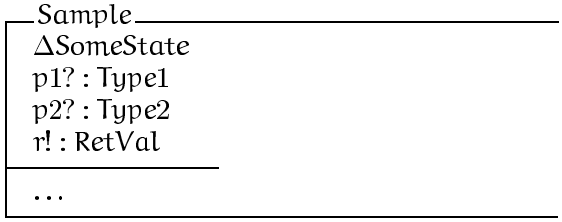
6 Translations of Z to TASPEC

The translation of Z to TASPEC is semi-automatic since the user must provide mappings between names in the Z specification and the given implementation. The translator contains information about Z data types such as sets, sequences, and functions and how they map into TASPEC. For example, if a Z type is a set, it is understood that $\cup$ can be used to add new elements.

We begin by presenting the technical details of the translation algorithm. Pseudo code for the algorithm is provided in Appendix A. After the technical discussion is an example of translation.

6.1 Translation Limitations

Translation is currently limited to accepting concrete refinements of Z as input. We restrict the use of such Z operators as $\lhd$ or $\rhd$ and $\nrightarrow$ or $\rightarrowtail$ as well as functions like $partition$ and rev. The assumption is that such operations wouldn't show up in a concrete refinement due to type limitations in the implementation language. This simplifies translation and seems reasonable from the refinements thus far examined.




```
Ret_Value SomeFunction(type_1 a, type_3 b,
        type_2 c);
```

$\langle SomeFunction \mapsto Sample, a \mapsto p1?, b \mapsto \varnothing, c \mapsto p2?, result \mapsto r! \rangle$

Figure 3: Information mapping a Z schema to a C function.

An additional limitation is that the user must provide mappings between names in Z and the implementation. Such information cannot be automatically determined in most cases [GHM87].

6.2 User Required Information

As noted above, the user must provide mappings for names in the specification to names in the implementation. Translation of state schemas requires mappings between schema declarations and program variables. The translator records type information about schema declarations and program variables. This type information is later used to find appropriate translation rules and generate bounds checking predicates for arrays.

Mappings are required for operational schemas to link schema and function names and match parameters. Figure 3 shows an example of the mapping information between a schema and a C function. Given are the schema declaration part and the function signature. Below that is a sequence which shows how names are mapped from Z to C. The first term links the function name to the schema name. Subsequent terms link the function parameters to schema input and output values. As shown, not all of the implementation parameters may be represented in the specification. The special value result maps function return values.

An operational schema need not map to a C function. In some cases, for example the TCP example in section 7, a schema can describe variable assignments. In that example, the mapping information indicates that a schema maps to an assignment operation. Translation of the schema results in location specifiers being linked to variable assignments.

6.3 Translation Rules

Predefined rules are used to translate Z expressions to TASPEC. The translation rules are divided into two kinds: data-type translation and operator translation. Data-type translation rules define how Z types map into TASPEC. The operator translation rules define the TASPEC query or state change resulting from a given Z operator.

The following describes how Z types are represented in TASPEC. Additional rules are described in [Hel95].

Sets are represented as assertable facts. For example, $s : \mathbb{P}\,Student$ becomes $s(x)$. If used in a matching expression, x is bound to some value of the set s. More complicated sets are represented by adding more parameters to the fact: $records : \mathbb{P}(Artist \times Title)$ becomes $records(a, t)$.

Relations/Functions are treated as a tuple with some constraint between elements. Given some function $fun : X_1 \times X_2 \rightarrow Y_1$, we match with $fun(x1, x2, y1)$. Values for the function would be asserted into the database and matches performed to get values for different parameter values.

Numbers are stored as single element sets. For example, a variable $max : \mathbb{N}$ is represented as $max(x)$. The translator will only assert one value at a time to the name max.

Operator translation rules allow simple Z expressions to be transformed into TASPEC. The translation algorithm will break a Z schema into a series of simple predicates. A simple predicate can be of the forms:

1. Expr op Expr or op Expr
2. v op Expr.

Case 1 represents preconditions and invariants. Case 2 represents variable definitions in postconditions. Each Expr may be composed of further simple predicates. Many of the translation rules have different forms depending on the form of the predicate. For example, the equality operator has a different translation in a variable definition than in a precondition.

The following are examples of operator translation rules:

Sets: In the examples, s_n is a set and e_n is an element.

$e_1 \in s_1$ and $e_1 \notin s_1$ become $s1(e1)$ and $\neg s1(e1)$.

$s_1 = s_2$ becomes $s1(e1) \cap s2(e1)$.

$s_1' = s_1 \cup e_1$ becomes **assert** $s1(e1)$.

Functions: $x_1' = x_1 \oplus \{y_1 \rightarrow y_2\}$ becomes **assert** $x1(y1, y2)$.

Numbers: $x' = x * y$ becomes
$x(e1) \cap y(e2)\{\textbf{retract}\ x(e1), \textbf{assert}\ x(e1 * e2)\}$.

$x < y$ becomes $x(e1) \cap y(e2) \cap e1 < e2$.

6.4 Required Definitions

The translator must be able to distinguish preconditions from definitions in schemas. Preconditions can only contain pre-names. Predicates that define new values contain a mixture of pre- and post-names. For example, $foo < bar$ is a precondition and $foo' = foo + 1$ is a definition, as is $foo' < bar$.

The order of evaluation of predicates in schemas is not always the order in which they appear in the schema. To successfully translate from Z to TASPEC, dependencies between predicates must be identified, and defining predicates potentially reordered, so as not to make use of values before they are defined. For a defining predicate to be evaluated, all free variables in E (denoted Var[E]) must be defined.

A sequence of defining predicates $A = \langle v_1 := E_1, v_2 := E_2, \ldots, v_n := E_n \rangle$ are termed "well-ordered" ([Jia95]) with respect to the pre-names of the schema if each v is distinct and the variables in each E are only those with previously defined values. Each defining predicate can proceed only when all free variables in E_i have been previously defined either by starting in the set of pre-names or by being given a value by a previous defining predicate.

Predicates in a schema may be connected by conjunctions and disjunctions. By default, a series of predicates listed without any logical connectives between them are connected by implicit conjunctions. Predicates connected by disjunctions result in multiple paths through a schema. The translator will generate separate location specifiers for each distinct "branch" of predicates.

6.5 Translation Algorithm

Translation of state schemas is handled separately from operator schemas due to differences in user required information and different kinds of mappings from specification to code. The different algorithms are discussed below.

Operator Translation

An overview of the operator translation algorithm is as follows:

1. Break a complex predicate into simple predicates. Operators connecting simple predicates are stored. Translation of simple predicates proceeds in order of operator precedence.

2. Given a simple predicate, identify the operator and choose the correct translation rule.

3. Generate names for new identifiers, apply the translation rule, and output the translated predicate.

State Schema Translation

This translation algorithm takes declarations and invariants in a state schema and translates into TASPEC. The user supplies mappings between declared names in the state schema and global variables in the implementation. This information allows arrays to be identified so that bounds checking predicates can be generated.

The following is an outline of the translation steps:

1. The translator requires name mapping information, the schema declaration list, and the predicate list: *UserInfo*, *DeclList*, and *InvList*.

2. For each declaration in *DeclList*, identify and record the Z type and generate any required initializations for this type (required for sequences). If a declaration maps to an array, generate array bounds checking predicates.

3. For each invariant in *InvList*, perform operator translation as previously described.

Operator Schema Translation

This translation algorithm takes an operational schema and translates to TASPEC. A part of the translation algorithm is inspired by the EX algorithm presented in [Jia95]. Translation is performed in two parts: preparation and translation. Both algorithms are shown in Appendix A.

The algorithm *PrepareSchema* is required to take a schema and return a list of precondition predicates and assignment predicates. The algorithm accepts a conjunctive list of predicates *Pred* that represent one possible path through the schema. Additionally, the list of pre-names and post-names must be provided. The assignment sequence that is returned will be correctly ordered.

The algorithm *TranslateSchema* takes name mapping information from the user, the list of preconditions, and the correctly ordered list of assignments and generates a TASPEC specification.

The overall translation algorithm is as follows:

1. The user chooses a Z operator schema to translate and constructs the appropriate information (Section 6.2) stored into UserInfo.

2. The chosen schema is expanded to include all schema inclusions (except state schemas) and resolve schema operations such as disjunction and conjunction.

3. Set *PreNames* to the pre-names from the declaration part and PostNames to the post-names. Convert the predicate part of the schema, *Pred*, to disjunctive normal form.

4. Call PrepareSchema(P, PreNames, PostNames, PreCond, AssignSeq) for each element P in the sequence *Pred*.

5. Call TranslateSchema(UserInfo, PreCond, AssignSeq) for each *PreCond* and *AssignSeq* assigned from the previous step.

The Z specification for this example (see figure 4) is taken from [Wor92]. The example is of a database to keep track of whether students are enrolled in a class and whether they have taken a particular test. Only a representative portion of the example will be translated.

[Student]

$size : \mathbb{N}_1$

ClassC

$sarray : (1 \,..\, size) \rightarrow Student$
$ectr : 0 \,..\, size$
$tctr : 0 \,..\, size$

$tctr \leq ectr$
$\forall i, j : 1 \,..\, ectr \mid i \neq j \bullet sarray\ i \neq sarray\ j$

EnrolokC

ΔClassC
$s? : Student$
$r! : Response$

$\forall i : 1 \,..\, ectr \bullet sarray\ i \neq s?$
$ectr < size$
$tctr' = tctr$
$ectr' = ectr + 1$
$sarray' = sarray \oplus ectr' \mapsto s?$
$r! = success$

Figure 4: State schema for Class Enrollment example.

An array holds the students with partitions of that array indicating enrolled and tested students. The invariants show that there should not be more tested than enrolled students and that all enrolled students are unique.

The translation of Figure 4 proceeds as follows. The state schema is mapped to the implementation by: $\langle student_list \mapsto sarray, e_ctr \mapsto ectr, t_ctr \mapsto tctr \rangle$. The constant size is a predefined number. sarray is identified as an array and has bounds checking schemas generated for it. TASPEC code generated for the predicate part will be:

assert $size(initial_value)$;
location decl student_list[initial_value]
{**assert** $array(student_list, 0, initial_value)$; }
location variable student_list[i]
{**assert** $arrayref(student_list, i)$; }
$arrayref(a, i) \cap array(a, l, u) \Rightarrow i \leq u \cap i \geq 1$;
$ectr(x) \cap 0 \geq x \cap size(y) \cap x \leq y$;
$tctr(x) \cap 0 \geq x \cap size(y) \cap x \leq y$;

The array bounds checking will be triggered by the array declaration and any reference to the array. Checking is also included for the legal ranges of ectr and tctr.

The first invariant is translated using a standard rule for numbers. The second invariant relies on TASPEC using implicit universal quantification for database matches (i.e. each assert will trigger all appropriate predicates associated with the asserted element). The translation uses a standard rule for universal quantification, inequality, and function evaluation. The result is:

$ectr(x) \cap tctr(y) \cap x \leq y$
$sarray(i, s) \cap sarray(j, t) \cap i \neq j \cap s \neq t$

Translation of EnrolokC requires the following user information: $\langle enrol_ok \mapsto EnrolokC, stud \mapsto s?, result \mapsto r! \rangle$. The result of *PrepareSchema* will be a *PreCond* list containing the first two predicates in the EnrolokC declaration list and a *AssignSeq* list with the remaining predicates in order as given. There is only one path through EnrolokC so *PrepareSchema* is only called once. *TranslateSchema* performs operator translation on the two lists output from *PrepareSchema* and results in the following specification:

location $enrol_ok(s)$**result** success
$\{(\neg sarray(i, s) \cap i \geq 1 \cap ectr(x) \cap i \leq x) \cap$
$(size(y) \cap x < y)\{$
retract $ectr(x)$,
assert $ectr(i + 1)$,
assert $sarray(i + 1, s)\}\}$

7 Example: TCP

This example examines how the previous techniques can be applied to testing the TCP protocol. TCP/IP is the de-facto standard for Internet communications. Implementations of TCP have several known security flaws that have been well documented elsewhere [Bel89]. This example examines the detection of a new security related bug in the TCP driver for 4.4BSD-Lite [Net93].

The TCP protocol is specified as a finite state machine. Each state in the protocol represents a stage in the process of opening, closing, or using a network connection.

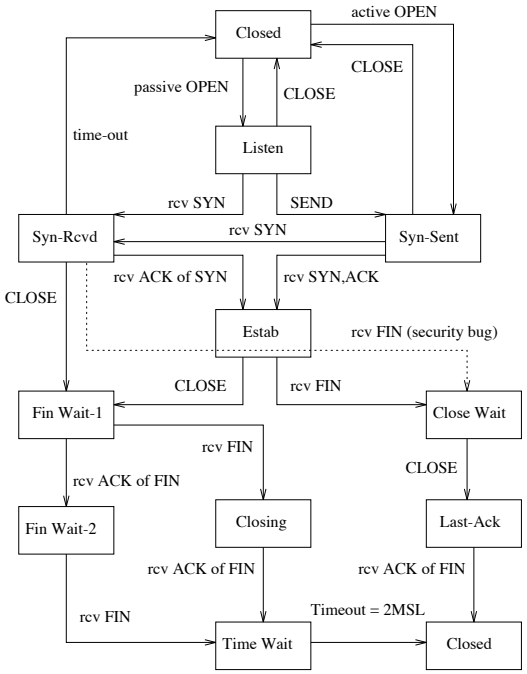


Figure 5: TCP State Transition Diagram

Transitions between states occur based on the types of messages sent and received during communication. For the purposes of testing the protocol for flaws, it is necessary to find if illegal transitions between states are possible in a given implementation. We will accomplish this by specifying the TCP state machine in Z, translating the specification to TASPEC, and then by applying automated testing tools driven by the TASPEC specification. This testing will demonstrate the existence of a newly discovered bug in the 4.4BSD-Lite source code [Guh95].

7.1 Z Specification for TCP Protocol States

This example takes a partial Z specification of TCP and uses it to drive the testing process. Our goal in testing has been finding improper state transitions in the protocol. This goal removed the need for a full specification (such as described in [GJ]) which is a great time savings when only a part of a system need be tested. Since only illegal TCP state translations are being tested for, only the TCP state machine is specified (see Figure 5).

A specification of this in Z is straightforward and illustrated in Figure 6. First the different states are enumerated. Next defined is the TCP state schema that holds the current state. Finally, a series of schemas specifying how each state is entered is defined. A state is potentially entered only from a set of legal prior states as illustrated in Figure 5.

$$TCP_State ::= Closed \mid Listen \mid Syn_Sent \mid Syn_Received \mid \ldots$$

TCP

$$state : TCP_State$$

Enter_Listen

$$\Delta TCP$$

$$state = Closed$$
$$state' = Listen$$

Enter_Syn_Received

$$\Delta TCP$$

$$state = Listen \lor state = Syn_Sent$$
$$state' = Syn_Received$$

Figure 6: Partial Specification of TCP in Z.

7.2 Translation of TCP Specification

To perform the translation, the TCP implementation must be examined. The TCP driver manages state transitions by storing the current state in the "t_state" field of the "tcpcb" structure. State changes are accomplished by storing a new value in the "t_state" field.

Translation requires the following user information for each state transition schema: $\langle assignment \mapsto Enter_Listen, tcpcb.t_state \mapsto state \rangle$. Translation proceeds as described in the previous section with the resulting specifications:

location assign tcpcb.t_state = Listen
    {state(Closed)
        {**retract** state(Closed),
            assert state(Listen)}}
location assign tcpcb.t_state = Syn_Sent
    {state(Closed)
        {**retract** state(Closed),
            assert state(Syn_Sent)}}
...

7.3 Testing TCP

The specification in Section 7.2 identifies assignments to tcpcb.t_state as significant (it tracks the TCP state) and worth testing. The use of state(Closed) as a predicate indicates that prior definitions of tcpcb.t_state are of interest (since that is where the state element is raised.) Therefore, the coverage requirements with respect to this specification are pairs of potentially sequential assign-

ments to that variable (determined by syntactically valid paths through the program).

The test oracle keeps track of the TCP abstract state, through the **retract** and **assert** mechanisms. When a state transition to Listen or Syn_Sent is made, the execution monitor checks the abstract state. If the state element state(Closed) is not present, an error condition is raised.

Such an approach finds an illegal transition from the Syn-Received state to the Close-Wait state. As shown in Figure 5, receiving a FIN packet in the Syn-Received state will cause an illegal transition to Close-Wait. This transition will be detected because the coverage requirements are to test all pairs of sequential assignments to tcpb.t_state. Any illegal pairs that are tested and succeed will be detected as errors.

8 Conclusion

TASPEC is a step towards providing a translation level between specification languages like Z to implementation languages like C. Currently, test oracles and coverage criteria are created automatically from **TASPEC**. Eventually other testing tools will also be developed to use the **TASPEC** notation. The advantage of using a language like **TASPEC** as a communication with testing technology is that it is specification language independent. Currently, translation from Z to **TASPEC** is available. Future work includes automation of the Z translation and investigation into translations of other widely-used specification languages such as Larch and VDM.

References

[AI91] Derek Andrews and Darrel Ince. *Practical Formal Methods with VDM*. McGraw-Hill, 1991.

[Bel89] S. M. Bellovin. Security problems in TCP/IP protocol suite. *Computer Communications Review*, April 1989.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 2nd edition, 1984.

[CRS] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. Submitted to ISSTA 1996 as a Regular Paper.

[DF93] Jeremy Dick and Alain Faivre. *Automating the Generation and Sequencing of Test Cases from Model-Based Specifications*, chapter 4, pages 268–284. First International Symposium of Formal Methods Europe Proceedings. Springer-Verlag, 1993.

[Dil90] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1990.

[Fin95] George Fink. *Discovering security and safety flaws using property-based testing*. PhD thesis, UC Davis, 1995.

[FL94] George Fink and Karl Levitt. Property-based testing of privileged programs. In Bob Werner, editor, *Tenth Annual Computer Security Applications Conference*, pages 154–163. IEEE Computer Society Press, December 1994.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(12):156–173, June 1975.

[GH93] John V. Guttag and James J. Horning. *Larch: Langauges and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GHM87] John D. Gannon, Richard G. Hamlet, and Harlan D. Mills. Theory of modules. *IEEE Transactions on Software Engineerin*, SE-13(7):820–829, 1987.

[GJ] Joshua D. Guttman and Dale M. Johnson. Three applications of formal methods at MITRE. Unpublished Notes.

[Guh95] Biswaroop Guha. Vulnerability analysis of TCP/IP suite. Master's thesis, University of California at Davis, September 1995.

[Hel95] Michael Helmke. A semi-formal approach to the validation of requirements traceability from Z to C. Master's thesis, UC Davis, September 1995.

[How75] William E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, C-24(5):554–559, May 1975.

[Jia95] Xiaoping Jia. An approach to animating Z specifications. In *Proceedings of the 19th Annual International Computer Software and Applications Conference*, Dallas, Texas, August 1995.

[Kin92] Steve King. *Specification Case Studies*, chapter The use of Z in the restructure of IBM CICS, pages 202–213. Prentice Hall, New York, New York, 1992.

[Net93] Netbsd, 1993. Source code for the TCP network device driver available at ftp://ftp.cdrom.com.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *14th International Conference on Software Engineering*, May 1992.

[Ric94] Debra Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994.

[San89] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs*. PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[SH94] S. Sankar and R. Hayes. Adl — an interface definition language for specifying and testing software. Technical Report CMU-CS-94-WIDL-1, Carnegie-Mellon University, January 1994.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, London, 2nd ed. edition, 1992.

[Wor92] John B. Wordsworth. *Software Development with Z*. Addison-Wesley, Workingham, England, 1992.

A Translation Algorithms

Algorithm A.1

```
proc PrepareSchema(in Pred, PreNames, PostNames,
  out PreCond, AssignSeq) ≡
  PreCond :=
      {p : Pred | p involves only PreNames};
  Definition := {p : Pred | p is definitive};
  AssignSeq := ⟨⟩;
  DefNames := ∅;
  done := false;
  while ¬done do
      progress := false;
      for d ∈ Definition do      d is of the form v = E
          if Var[E] ⊆ DefNames ∪ PreNames
            then
                Definition := Definition \ {d};
                DefNames := DefNames ∪ {v};
                AssignSeq := AssignSeq ⌢ ⟨v := E⟩;
                progress := true;
          fi
      od
      done := ¬progress ∨ Definition = ∅ ∨
      DefNames = PostNames;
  od
end
```

Algorithm A.2

```
proc TranslateSchema(in UserInfo, PreCond,
      AssignSeq) ≡
  if UserInfo indicates a translation to a function
    then
        output "location funcall <FcnName>",
            "result <ResultName>";
        if UserInfo indicates a return value for
          the function AND return value is assigned
          a constant value in AssignSeq
          then
              rval := {e : AssignSeq | variable assigned
                  to is return value};
              AssignSeq := AssignSeq \ {rval};
              output "if <rval assignment",
                  "variable> = <rval constant",
                  " assignment>";
        fi
        for p ∈ PreCond do
            p_tas := translate p to TASPEC;
            output "<p_tas>";
            connect each consecutive precondition
                with "AND";
        od
        for a ∈ AssignSeq do
            a_tas := translate a to TASPEC;
            output "<a_tas>";
            connect each consecutive assignment
                with",";
        od
  elsif UserInfo indicates a translation to assignment
      then
          output "location assign <VarName> =",
              "<AsgnValue>";
          if p ∈ PreCond guards assignment
            then
                p_tas := translate p to TASPEC;
                output "if <p_tas>"
          fi
          a_tas := translate a ∈ AssignSeq to TASPEC;
          output"<a_tas>";
  fi
end
```