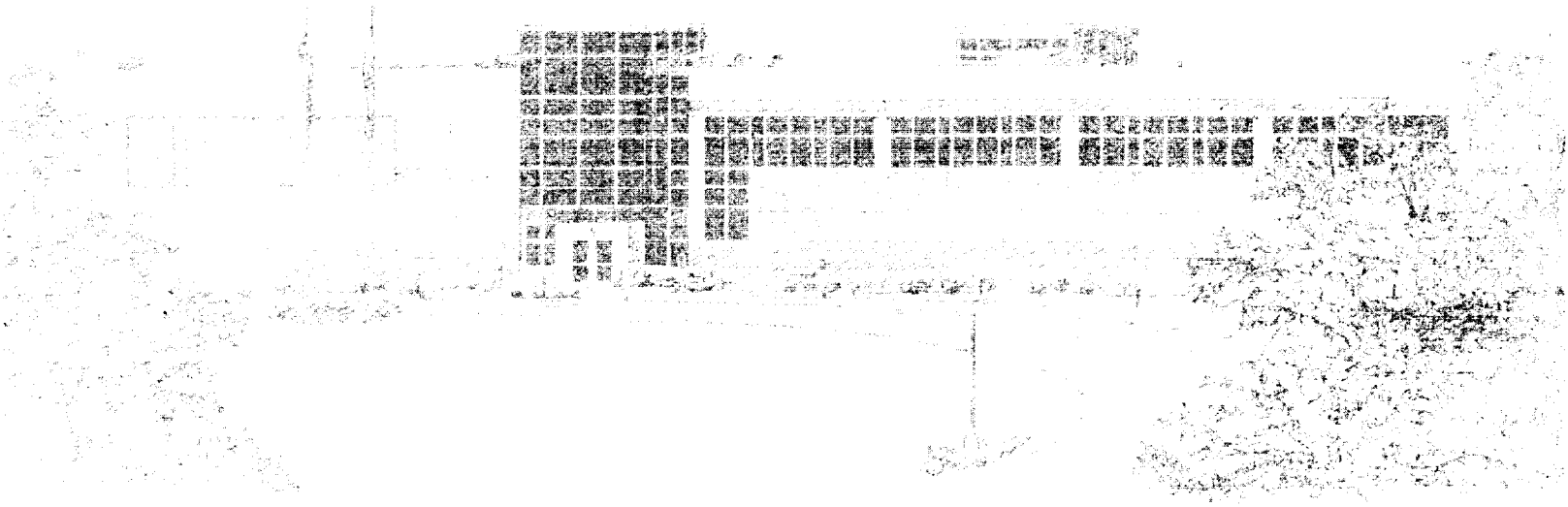
NASA-IVV-97-012
WVU-IVV-97-012
WVU-CS-TR-97-015

NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

Generating Test Templates via Automated Theorem Proving

By Mani Prasad Kancherla



National Aeronautics and Space Administration

West Virginia University

NASA-IVV-97-012

NASA IV&V Facility, Fairmont, West Virginia

Generating Test Templates via Automated Theorem Proving

Mani Prasad Kancherla

September 3, 1997

This technical report is a product of the National Aeronautics and Space Administration (NASA) Software Program, an agency wide program to promote continual improvement of software engineering within NASA. The goals and strategies of this program are documented in the NASA software strategic plan, July 13, 1995.

Additional information is available from the NASA Software IV&V Facility on the World Wide Web site http://www.ivv.nasa.gov/

This research was funded under cooperative Agreement #NCC 2-979 at the NASA/WVU Software Research Laboratory.

Generating Test Templates via Automated Theorem Proving

THESIS

Submitted to the Eberly College of Arts and Sciences
Of
West Virginia University

in partial fulfillment of the requirements for
the degree of Master of Science

by
Mani Prasad Kancherla
Department of Statistics and Computer Science
West Virginia University

Approval of Examining Committee

Dr. Steve Easterbrook, Ph.D.

Dr. James D. Mooney, Ph.D.

Date

Dr. John R. Callahan, Ph.D. (chair)

Acknowledgements

This work would not have been possible without the support of Dr. Callahan. I gratefully acknowledge his valuable assisstance and encouragement. I also thank other committee members Dr. James D. Mooney and Dr. Steve Easterbrook for their insightful comments. Special thanks are due to Edward Addy, Frank Schneider and other SRL team members for their suggestions.

I would also like to thank the people at NASA/WVU Research Lab, Concurrent Engineering Research Center, and the Department of Statistics and Computer Science for their cooperation.

Lastly, but most importantly, I would like to thank my parents, my brother and sister for their unquestionable support, love and encouragement.

Contents

Contents

Abstract

Testing can be used during the software development process to maintain fidelity between evolving specifications, program designs, and code implementations. We use a form of specification-based testing that employs the use of an automated theorem prover to generate test templates. A similar approach was developed using a model checker on state-intensive systems. This method applies to systems with functional rather than state-based behaviors. This approach allows for the use of incomplete specifications to aid in generation of tests for potential failure cases. We illustrate the technique on the cannonical triangle testing problem and discuss its use on analysis of a spacecraft scheduling system.

1. Introduction

The major limitation of conventional testing is that it can only show the presence of errors but never their absence[1]. This is because we usually have an infinite (or very large) Input Space and testing over all possible values of input is impractical.

Testing is a process of verifying whether a program does what it is supposed to do. In other words, a program is correct if it meets its requirements. Typically software requirements will be specified in a natural language and can be translated into a set of properties that the software (or program) should exhibit. We claim that a program is partially correct if it exhibits all the properties stated in the requirements specification. While it is possible to specify a program in a formal specification language and verify whether the specification exhibits the required properties or not, its usefulness is limited for the following reasons:

Inconsistencies between the formal specification and the program: There are a number of reasons why the actual program and the formal specification can be inconsistent. One possibility is that the specification was developed at an early stage in the life cycle of the software and the changes made in the later phases are not reflected in the formal specification. Hence, proving that the formal specification exhibits a property does not necessarily mean that the actual program exhibits that property.

Partial formal specification/verification: It is usually very expensive (and often unnecessary) to specify a huge program completely in a formal specification language and prove its correctness. We can specify only the critical sections in a formal language and verify the partial specification. Properties exhibited by the partial specification do not necessarily mean that the program will exhibit them because of the inadequacy in detail.

Non-functional requirements: There could be a number of non-functional requirements that should be exhibited by the program. For example, there could be performance constraints on the program. These properties are implementation specific and it is usually inappropriate to prove that the specification exhibits these properties.

For the above reasons, we not only want to verify the functional correctness of the specification but we would also want to generate test cases so that we can verify the actual program for correctness. In this thesis, we propose a method of generating test templates for each of the functional properties specified in the requirements specification. A test template can be thought of as a set of conditions on the Input Space. Testing the software for any one particular instance of the template is minimally necessary to prove that the actual program exhibits the corresponding property.

The remainder of this thesis is organized as follows: Chapter 2 discusses the related work. Chapter 3 presents a simple example based on Myers cannonical triangle example that will demonstrate our approach to test template generation. Chapter 4 describes the strategies for

deriving test templates and structuring them into Test Template Hierarchy (TTH). Chapter 5 provides a detailed example based on the modified triangle specification. Chapter 6 presents a practical example and discusses the usefulness of our approach in the real world. Chapter 7 presents an overview of this thesis and concludes with the scope for future work. Appendix A presents an introduction to Prototype Verification System (PVS). Appendix B presents the incorrect PVS model for the modified triangle specification. Finally, Appendix C presents the corrected model for the modified triangle example and the generated test templates.

2. Related Work

Phil Stocks and David Carrington in their paper "A framework for specification based testing" [2] suggest a method for deriving test templates from Z-specification and provide a test template framework for structuring the tests. They define the Input Space (IS) of an operation as the space from which input can be drawn, i.e. IS represents type-compatible input to the operation. The Valid Input Space (VIS) is the subset of IS for which the operation is defined. They claim that all the tests for an operation must be derived from the operation's VIS because the specification defines only what happens for input in the VIS. Once the VIS of an operation is determined, they subdivide the VIS into subsets called domains by applying testing strategies and heuristics. A template hierarchy is constructed with the templates as nodes and strategies as edges. After applying all the desired strategies, each instance of a terminal template in the hierarchy graph is considered equivalent to all other instances of this template for testing purposes. In their approach it is not clear how the generated test templates relate to the properties stated in the requirements. Hence it is not evident whether the generated test templates are sufficient to test for all the properties stated in the requirements.

The formal specification language, Maribila [3], was designed by Computationl Logic, Inc (CLI) and the Open Group Research Institute (RI) with the goal to make it easy for

software engineers to read and write Maribila formal specifications without specific formal methods expertise. Maribila has a formally defined semantics. It is syntactically and semantically equivalent to C++ or Java, but has language features that encourage abstraction. Maribila formal specifications can be used to drive system testing and the technique is called *specification-based evet-trace testing.*

In the event-trace testing methodology, the system architecture is formally described as an abstract program in the formal language Maribila. The Maribila abstract program specifies that the system will take certain actions, and constrains acceptable orderings of these actions. The abstract program steps can be viewed as significant events that must take place in the course of system execution. The abstract program defines a finite state machine that will accept or reject an event trace. An event trace is a stream of event occurences. A finite state machine is created for each system interface specified in Maribila by a prototype tool, *acceptor generator.* The tool identifies the events suggested by the specification, and defines a program interface for announcing the events from the system code base at run time. The programmer instruments the code base to emit the appropriate event announcements and ensures that the expected events are announced in a correct order. The instrumentation also records which states of the finite state machine have been visited, giving a metric for coverage of the test suite.

Daistish [4], a tool developed by Merlin Hughes and David Stotts, creates effective test drivers for programs in languages that use side effects to implement Abstract Data Types

(ADTs). The tool performs systematic algebraic testing. The basic approach is to select appropriate data points (values for parameters to the operations called in axioms), compute the right and left sides of an axiom separately, and then compare the results. A correct implementation should produce values for each side that are equivalent.

Diastish is a Perl script which processes a formal specification of an ADT along with the code for an object implementing the ADT, to produce a test driver. The specification files contain axioms and test vectors (sample instantiations of types used by the axioms). Daistish scans all specification files and code is produced to instantiate each test vector and evaluate each axiom. The axioms are then called with each possible valid combination of parameters available from instantiations of the test vectors. If an axiom fails, test generator will output the axiom name that failed and the names of data points used as parameters. Otherwise statistics are collected for each axiom and summarized at completion.

Bruno Dutertre and Victoria Stavridou describe the application of a formal approach to the specification and analysis of a safety critical system in their work "Formal Requirements Analysis of an Avionics Control System" [5]. Their work is based on an Air Data Computer (ADC) that consisted of two channels. A primary channel performs all ADC functions during normal operation and a backup channel takes over when the primary fails. The functional requirements were specified and verified using the Prototype Verification System (PVS).

3. A Simple Example

In this section we'll illustrate test template generation with a simple example. Consider the following specification:

> *Given three integers representing the three edges of a triangle, determine the type of the triangle (i.e. Equilateral, Isosceles or Scalene). If all sides of the triangle are equal then it is Equilateral. If two sides are equal and the third side is different from them then it is Isosceles. If all sides are different then it is Scalene." [6]*

As the specification clearly states that the 3 integers represent the edges of a triangle, we don't need to test whether the 3 integers form a valid triangle. Note that it is also implied in the above specification, that any triangle will be one of the 3 types specified. We'll come back to this specification in chapter 5 and generalize it to handle inputs that do not form a triangle.

In the PVS specification given in Figure 1, we declare x, y, and z as variables of type POSITIVE INTEGER. We also declared an enumeration type named 'Triangle_type' having values 'Scalene', 'Isosceles', 'Equilateral' and 'Error'. Note that even though any triangle will be one of the 3 types Equilateral or Isosceles or Scalene, we added 'Error' in

our enumeration type because it is useful in proving certain properties. The function 'Triangle' accepts 3 positive integers as parameters and returns the type of the triangle. Note that the function 'Triangle' will return "Error" if the triangle is not Equilateral or Isoceles or Scalene.

An attempt to prove the property "It is never the case that the triangle is not equilateral or isosceles or scalene" yields the following two proof goals*:

proof goal 1: Given x, y, and z are integers and y = z, prove that x = z.

proof goal 2: Given x, y, and z are integers and x = z, prove that x = y.

The above proof goals could not be proved since the statements **(y = z) => (x = z), (x = z) => (x = y)** for any three positive integers x, y, and z could not be shown. So we conclude that the PVS specification shown in Figure 1 will not satisfy the above stated property. We can generate the following test templates from the above proof goals†:

Test template-A: x, y, and z are positive integers and (x, y, z) form a triangle and

(y = z) and not(x = z)

Test template-B: x, y, and z are positive integers and (x, y, z) form a triangle and

(x = z) and not(x = y)

---

* There will be three more proof goals generated by PVS that are automatically proved by the PVS theorem-prover. Only the two proof goals mentioned here could not be proved automatically.

† The actual strategy for generating test templates is described in chapter 4.

```
triangle : THEORY
 BEGIN
        x, y, z: VAR pos
        Triangle_type: TYPE = {Scalene, Isosceles, Equilateral, Error}

        Triangle(x, y, z): Triangle_type =
                IF x = y AND y = z THEN Equilateral
                ELSIF x = y AND z /= y THEN Isosceles
                ELSIF x /= y AND y /= z AND z /= x THEN Scalene
                ELSE Error
                ENDIF

        Conj: CONJECTURE Triangle(x, y, z) /= Error
 END triangle
```

Figure 1: Incomplete specification of Triangle problem in PVS

Note that "***(x, y, z) form a triangle***" means that the three positive integers x, y, and z when interpreted as representing the lengths of the sides, form a triangle.

These test templates correspond to the cases where the specification (given in Figure 1) fails to exhibit the property. Note that three integers (when interpreted as representing the lengths of sides) form a triangle if the sum of any two is greater than the third.

Instances of the template-A consist of all the 3 integer tuples of the form (x, y, y) where x and y are two different positive integers and (x, y, y) form a triangle. i.e. Instances of template-A consist of the infinite set

{ (1,2,2), (1,3,3), (1,4,4), ...

(2,1,1), (2,3,3), (2,4,4), ..

....}

Similarly, the instances of template-B will consist of all the 3 integer tuples of the form (x, y, x) where x and y are two different positive integers. Instances of template-B consist of the infinite set

{ (1,2,1), (1,3,1), (1,4,1), ...

(2,1,2), (2,3,2), (2,4,2), ...,

.....}

Note that these two test templates correspond to the case of Isosceles triangles (in both the cases we have two equal sides and a different third side). As this is a very simple example, you can see that in the model (i.e., PVS specification) we did not consider all the possible cases for Isosceles triangle. All the possible cases of two equal sides and a different side would be:

((x = y) AND (y /= z)) OR
((y = z) AND (x /= z)) OR
((x = z) AND (x /= y)).

But we specified only the condition ((x = y) AND (y /= z)) in our model and did not specify the other two conditions. So our model was incorrect. We generated test templates corresponding to these two cases. The specification after fixing the above error is given in Figure 2.

The above mentioned property can be easily proved based on the corrected specification. Now, we can try to prove the other 3 properties:

1. If all sides are equal then Equilateral.
2. If two sides are equal and third side is different then Isosceles.
3. If no two sides are equal then Scalene.

Since this is a very simple example and the above properties are trivially true for the model in Figure 2, the generated test templates will be no more than the specified conditions in each of the conjectures. Hence, the test templates corresponding to the above 3 properties will be:

Template-1: x = y AND y = z (x, y, and z are integers)

Template-2: ((x = y) AND (y /= z)) OR ((y = z) AND (x /= z)) OR

((x = z) AND (x /= y)) (x, y, and z are integers)

Template-3: x /= y AND y /= z AND z /= x (x, y, and z are integers)

```
triangle : THEORY
 BEGIN
        x, y, z: VAR pos
        Triangle_type: TYPE = {Scalene, Isosceles, Equilateral, Error}

        Triangle(x, y, z): Triangle_type =
        IF x = y AND y = z THEN Equilateral
        ELSIF ((x = y) AND (y /= z)) OR
                ((y = z) AND (x /= z)) OR
                ((x = z) AND (x /= y)) THEN Isosceles
        ELSIF x /= y AND y /= z AND z /= x THEN Scalene
        ELSE Error
        ENDIF

        conj: CONJECTURE Triangle(x, y, z) /= Error
        conj1: CONJECTURE (x = y AND y = z) IMPLIES
                Triangle(x, y, z) = Equilateral
        conj2: CONJECTURE ((x = y) AND (y /= z)) OR ((y = z) AND (x /= z)) OR
                ((x = z) AND (x /= y)) IMPLIES Triangle(x,y,z) = Isosceles
        conj3: CONJECTURE (x /= y AND y /= z AND z /= x) IMPLIES
                Triangle(x,y,z) = Scalene
 END triangle
```

Figure 2: Corrected specification of the 'triangle' problem in PVS

Each of these templates consists of an infinite set of instances. Any one particular instance of a template is sufficient to prove that the software exhibits a particular property. The actual procedure for generating test templates from the proof goals is described in the next chapter.

4. Heuristic Approach to Test Template Generation

We suggest a new method of testing software based on the formal specification. We used the Prototype Verification System (PVS) and its in-built theorem prover to derive test templates corresponding to the properties stated in the requirements. After developing the PVS specification, we specify the properties stated in the requirements as conjectures. Using the theorem prover we try to prove that the conjecture is TRUE, i.e. we'll prove that the property holds for the PVS specification.

4.1 Proof Trees

PVS proof checker provides a collection of proof commands that can be combined to form proof strategies. Applying proof commands in order to prove a conjecture might yield:

1) another proof goal that needs to be proved in order to prove the original proof goal.
2) more than one proof goal. In which case, the proof is split into branches with sub goals. In order to prove the original proof goal we have to prove all the sub-goals.
3) termination of that proof branch in the case where the proof goal is trivially TRUE.

From the proof commands that are applied to the conjecture a proof tree is constructed where all the leaves in the proof tree are recognized as TRUE.

Consider the trivial example of specifying the function Division_Result that will return the type of integer division.

> *Given two integers x and y, return the type of 'x/y'. If 'y = 0' the function should return 'Error', otherwise it should return 'Positive' or 'Negative' or 'Zero' depending on the value of 'x/y'.*

The PVS specification for the above function is given in Figure 3. The function Division_Result accepts two integer parameters x and y. If 'y = 0' then the function returns "Error". If 'x = 0' (and y /= 0) the function returns "Zero". If both x and y are positive or negative then the function returns "Positive" otherwise it returns "Negative".

In the conjecture conj1 of Figure 3, we try to prove that the function Division_Result will not return "Error" if 'y /= 0'. The proof tree corresponding to this conjecture is depicted in Figure 4. The nodes of the proof tree are numbered and the proof command applied at each node is also shown.

```
example1 : THEORY
BEGIN

    x, y: VAR int
    Return_type: TYPE = {Positive, Negative, Zero, Error}

    % Returns the type of x/y
    Division_Result(x, y): Return_type =
        IF y = 0 THEN Error
        ELSIF x = 0 THEN Zero
        ELSIF (x > 0 AND y > 0) OR (x < 0 AND y < 0) THEN Positive
        ELSE Negative
        ENDIF

    conj1: CONJECTURE not(y = 0) IMPLIES Division_Result(x, y) /= Error

END example1
```

Figure 3: PVS specification of Division_Result function

Each node of the proof tree is a proof goal. Each proof goal has a sequent consisting of a sequence of formulas called antecedents and a sequence of formulas called consequents. In PVS, such a sequent is displayed as*

```
{-1} A1

{-2} A2

[-3] A3

     :

     :
```

* The antecedents are assigned negative numbers and the consequents are assigned positive numbers. The braces surrounding the number indicate that the formula has changed from the parent sequent. The square brackets surrounding the number indicate that the formula is repeated from the parent sequent.

```
      |--------
{1} B1
{2} B2
{3} B3
        :
        :
```

The sequent formulas Ai are the antecedents and the Bj are the consequents. The interpretation of a sequent is that the conjunction of antecedents should imply the disjunction of the consequents, i.e.

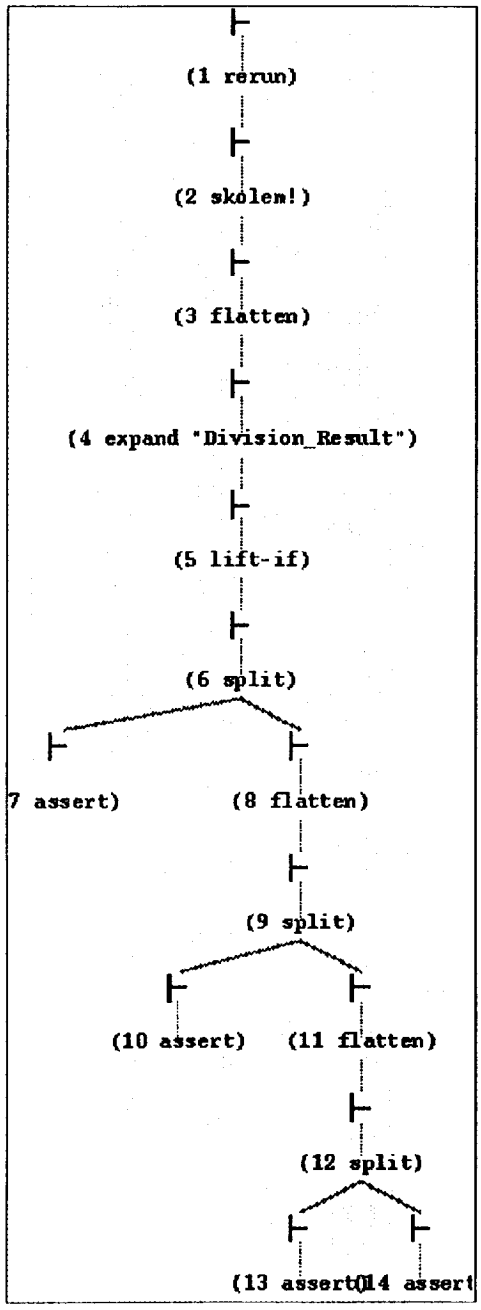


Figure 4: Proof tree for conj1 of Division_Result

(A1 ^ A2 ^ A3 ^ ...) --> (B1 V B2 V B3 V ...)

For example in the proof tree (shown in Figure 4) for conj1 of Division_Result at node 12 we have the following sequent

{-1} NOT((x!1 > 0 AND y!1 > 0) OR (x!1 < 0 AND y!1 < 0)) AND (Negative = Error)

|-------

[1] x!1 = 0

[2] y!1 = 0

[3] (y!1 = 0)

i.e., we have to prove that the following implication is TRUE (shown after replacing the skolemized variables with actual variables)

NOT((x > 0 AND y > 0) OR (x < 0 AND y < 0)) AND (Negative = Error) --> (x = 0) V (y = 0)

The above implication is TRUE since the condition (Negative = Error) is Flase on the left-hand side of the implication.

Note that the root of the proof tree is a sequent with the conjecture that we are trying to prove as the consequent and with no antecedents. PVS proof steps build a proof tree by adding subtrees to leaf nodes.

We claim that it is possible to generate test templates based on the proof tree that would test for the property corresponding to the conjecture we proved (or failed to prove). While generating test templates we need to consider the two cases:

1) when we fail to prove that a property is exhibited by the model

2) when we succeed in proving that the model exhibits a property.

4.2 Strategy for generating test templates from invalid proofs

If we fail to prove that a property is exhibited by the model, then there exist one or more proof goals in the proof tree that could not be proved to be TRUE, i.e. there are one or more sequents that are false.* We note that the implication of the form

(A1 ^ A2 ^ A3 ^ ...) --> (B1 V B2 V B3 V ...)

will be FALSE only when the left-hand side of the implication is TRUE and the right hand side of the implication is FALSE. Hence we have the condition

(A1 ^ A2 ^ A3 ^ ...) ^ NOT(B1 V B2 V B3 V ...)

---

* Note that the failure to prove a property does not necessarily mean that one or more sequents are false. It could also happen when we don't have enough information in the model to prove the conjecture. We are not interested in that case.

which will be the test template when we fail to prove the proof goal.

4.3 Strategy for generating test templates from valid proofs

To generate test templates when we succeed in proving a conjecture, the general approach is to look at the leaves of the proof tree. Each leaf is a sequent that is TRUE*. The implication

(A1 ^ A2 ^ A3 ^ ...) --> (B1 V B2 V B3 V ...)

will be TRUE if the left hand side is false or the right hand side is TRUE,

i.e., not(A1 ^ A2 ^ A3 ^ ...) V (B1 V B2 V B3 V ...)

i.e., not(A1) V not(A2) V not(A3) V ... V B1 V B2 V B3 V ...

The above condition is TRUE if one or more terms are TRUE. In general, if we have n terms, there are 2^n-1 ways the above condition could be TRUE. For each of those possibilities we will have a test template. For instance, in the above example, one test template would be

not(A1) ^ not(A2) ^ not(A3) ^ ... ^ B1 ^ B2 ^ B3 ^ ...

which corresponds to the case where all terms of the condition are TRUE. Another test template would be

(A1) ^ not(A2) ^ not(A3) ^ ... ^ B1 ^ B2 ^ B3 ^ ...

which corresponds to the case where all terms except the first term are TRUE. Note that some of these test templates will have no instances. For example, the test template

(x > 2) ^ not(x > 2)

has no instances.

In this approach we'll be generating a large number of test templates, exponential in the order of the number of sequent formulas (or conditions). Most of the test templates will have no instances because we are negating the antecedents that are usually the facts specified in the model[†]. We suggest a different approach that will generate fewer test templates. To understand the new approach, we need to understand how PVS generates proof goals when we try to prove a particular property.

---

[*] If a proof goal could not be proved (because the sequent is FALSE), then the "strategy for generating test templates from invalid proofs" should be applied.

[†] One or more of the antecedents could be wrong in the case where we are considering a branch of the proof tree that does not exhibit the property we are testing for.

Consider the conjecture conj1 of Division_Result. In conj1, we are trying to prove the property *"The function does not return Error when the divisor is non-zero"*. When we try to prove this property using PVS proof checker, PVS will check all possible execution paths through the program and compares the output of the model for each execution path with the symbol 'Error'. For the Division_Result example specification there are four execution paths through the program corresponding to the four possible outputs 'Error', 'Zero', 'Positive', and 'Negative'.

For instance, consider the sequent at node 10 of the proof tree shown in Figure 4:

{-1} x!1 = 0 AND (Zero = Error)

|-------

[1] y!1 = 0

[2] (y!1 = 0)

This sequent belongs to a proof branch that will yield the output 'Zero'. This proof branch satisfies the condition "divisor is non-zero" and returns the symbol 'Zero'. PVS was comparing the symbol 'Zero' with 'Error' to see whether they were equal. In which case, we will fail to prove the above property. But the condition 'Zero = Error' (which means that both the symbols "Zero" and "Error" are equivalent) is trivially FALSE. Since the left hand side of the implication is FALSE, the proof goal is TRUE.

Similar sequents will be generated when PVS explores the execution paths that would yield the ouput 'Positive' or 'Negative'. In either case the antecedent will be False and hence the proof goal will be TRUE. Now, we consider the execution path that yields the ouput 'Error' (i.e., the case where the divisor is zero) which corresponds to the sequent at the node 7 of Figure 4

[-1] y!1 = 0 AND TRUE

|-------

[1] (y!1 = 0)

In this case, we'll be able to prove the implication since the right hand side of the implication is a sub-condition of the left-hand side of the implication. The proof goal is TRUE since the consequent 'y = 0' appears as a part of the antecedent.

So we have the following two types of proof goals that are trivially TRUE.

1) The sequent has atleast one antecedent that is FALSE. In this case the left-hand side of the proof goal will be FALSE and hence the proof goal will be trivially TRUE.
2) One or more consequents also appear as antecedents. In this case, the right hand side of the implication is a sub-set of the left-hand side of the implication. So whenever the left-hand side of the implication is TRUE, the right hand side of the implication is also TRUE. Hence the proof goal is TRUE.

As we shall demonstrate, we can easily find out whether any of the antecedents are false. The false antecedents shall be ignored while generating the test templates. If we have one or more consequents that appear also as antecedents then they shall also be ignored. After removing the antecedents and consequents as explained above, the test template corresponding to a sequent of the form

(A1 ^ A2 ^ A3 ^ ...) --> (B1 V B2 V B3 V ...)

would be

(A1 ^ A2 ^ A3 ^ ...) ^ not(B1 V B2 V B3 V ...)

i.e.

A1 ^ A2 ^ A3 ^ ... ^ not(B1) ^ not(B2) ^ not(B3) ^ ...

However, this general approach will not always work. The reason being that there are a number of ways to prove a conjecture, and hence the proof tree for any conjecture is not unique. The test templates generated for different proof trees of the same conjecture differ in the detail they have. i.e., if we generate test templates based on two proof trees A and B; a single test template generated based on proof tree A might correspond to a number of test templates generated based on proof tree B. Note that in the case where we fail to prove a

proof goal, this not a problem because whatever the proof tree may be, it is sure to identify the case(s) for which the proof will fail.

Hence, we provide a heuristic approach to generating test templates for the case where we succeed in proving the conjecture. We generate a test template corresponding to every leaf of the proof tree as follows:

1. Prove the conjecture using only the fundamental rules (such as flatten and split). Do not use any strategies (such as 'grind' and 'ground').
2. Add all the conditions in the antecedents to the test template except those that are trivially FALSE.

   For example, in the Division_Result proof, consider the sequent at node 10:

```
{-1}   x!1 = 0 AND (Zero = Error)
 |-------
[1]    y!1 = 0
[2]    (y!1 = 0)
```

The condition 'Zero = Error' (which means that both the symbols "Zero" and "Error" are equivalent) is trivially FALSE. So we ignore that condition and add (x = 0) to the test template*.

3. If there are consequents, add all the negated consequents to the test template. Ignore the consequents all of whose conditions appear as antecedents.

   For example, in the Division_Result proof, consider the sequent at node 7:

```
[-1]   y!1 = 0 AND TRUE
 |-------
[1]    (y!1 = 0)
```

   In this case, the consequent [1] has only one condition (y!1 = 0) and that condition appears as part of the antecedent [-1]. So we ignore the consequent [1] while generating the test template.

   As another example, consider the sequent at node 13:

```
{-1}   ((x!1 > 0 AND y!1 > 0) OR (x!1 < 0 AND y!1 < 0)) AND (Positive = Error)
 |-------
[1]    x!1 = 0
[2]    y!1 = 0
```

* As we mentioned earlier, if the antecedent has two or more conditions joined by conjunction(s) then each such condition should be treated as a separate antecedent.

[3] (y!1 = 0)

In this case, the condition (Positive = Error) is False in the antecedent {-1}, so we ignore that condition and add the rest of the conditions of {-1} to the test template (refer to heuristic 2). None of the consequents appear as a part of any antecedent. So we negate and add all the consequents. Since the consequents [2] and [3] are the same, we can ignore [3].

4. Check whether the test template has enough conditions to define the input to the program. If yes we have the test template corresponding to the leaf. If not, consider the sequent immediately above that leaf in the same branch of the proof tree. If there is no such sequent, then the leaf cannot produce a test template. Otherwise repeat steps 2, 3, and 4 for this sequent. We repeat this process until we get enough conditions to clearly define the input to the program. There is no such case in our simple example Division_Result.

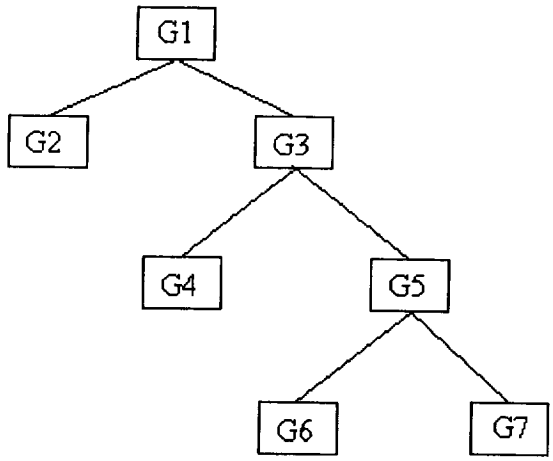


Figure 5: Skeleton of the TTH for conj1 of Division_Result

5. If no leaf of a subtree could produce a test template then the root of the subtree must be treated as a leaf. There is no such case in our simple example Division_Result.

6. If there is only one skolemized* variable corresponding to each original variable then replace the skolemized variables with the original variables, otherwise replace each skolemized variable with a unique variable name.

   For example, if the variable x appears as only one skolemized variable x!1, then replace the skolemized variable x!1 with x. If the variable y appears as two skolemized variables y!1 and y!2 then replace them with y1 and y2 respectively.

* Skolemization is a general technique to eliminate universal and existential quantifiers.

7. Add any additional constraints given in the requirements. In the Division_Result example, the additional constraints given in the requirements would be "x and y are integers".

4.4 Structuring the Test Templates

As we shall see from Figure 7, for a simple problem like the 'triangle' example we have a huge proof tree with approximately 50 leaves and each leaf lead to a test template. For a complex problem with lots of nested conditional statements, the number of test templates could easily become unmanageable. Hence we need to structure these test templates.

Since we generated test templates based on the proof tree, they have inherent hierarchy built into them. To arrange the test templates in hierarchy, remove all the nodes in the proof tree except the leaves and the nodes that join different branches of the proof tree. Then we will have a skeleton for the Test Template Hierarchy (TTH).

Consider the example PVS specification of Division_Result (refer to Figure 3). If we remove all the nodes except the leaves and the nodes that join different branches from the proof tree of conj1 (refer to Figure 4), we will have the skeleton of TTH for conj1. Skeleton of TTH for conj1 is depicted in Figure 5. 'G1' corresponds to the goal numbered 6 in the proof tree (refer to Figure 4). G2, G4, G6, and G7 in the TTH skeleton correspond

to the leaves numbered 7, 10, 13, and 14 in the proof tree respectively. The nodes G3, and G5 in the TTH skeleton correspond to the nodes 9, and 12 in the proof tree respectively.

To get the TTH, place each test template at the leaf that generated the template. If all the child nodes of a node have some condition in common then remove that condition from all the child nodes and put it at the parent node. Starting at the leaves repeat this procedure upto the root node. When we are done, we will have the TTH graph with conditions placed at the nodes.

The conjecture, conj1, of the PVS specification Division_Result (shown in Figure 3) yields the following test templates*:

T1: (y = 0)..... derived from the leaf no. 7 of proof tree

T2: (x = 0 ^ y /= 0)..... derived from the leaf no. 10 of proof tree

T3: (x > 0 ^ y > 0) V (x < 0 ^ y < 0) ^ (x /= 0) ^ (y /= 0)..... derived from the leaf no. 13

T4: not(x > 0 ^ y > 0) ^ not(x < 0 ^ y < 0) ^ (x /= 0) ^ (y /= 0).derived from the leaf no. 14

Now, we place T1 at the node G2 of TTH skeleton (refer to Figure 5), T2 at G4, T3 at G6, and T4 at G7. Since both the nodes G6 and G7 have the condition (x /= 0) ^ (y /= 0) in common, we move that condition to their parent node G5. Then we'll have the condition (x > 0 ^ y > 0) V (x < 0 ^ y < 0) at G6 and the condition not(x > 0 ^ y > 0) ^ not(x < 0 ^ y < 0) at G7, and the condition (x /= 0) ^ (y /= 0) at G5. Now, both the nodes G4 and G5 have the

* The actual procedure of deriving these test templates is exemplified in the next section.

condition (y /= 0) in common. So we move that condition to their parent node G3. Then we will have the condition (x > 0) at G4, and (x /= 0) at G5, and (y /= 0) at G3. Since G2 and G3 have no conditions in common we have finished building the TTH. The final TTH is shown in Figure 6.

To get back a test template from the TTH, we select a path from the root node of the TTH graph to a leaf and take the conjunction of all the conditions that appear in that path. Arranging test templates in this manner gives us the convenience of choosing test templates with a specific property.

In the above example, if we are interested only in the test cases that correspond to non-zero input. Then we have to look for the conditions (x /= 0) and (y /= 0). So, the path from the root node of TTH, G1, should include both the nodes G3 and G5, since G3 has the condition (y /= 0) and G5 has the condition (x /= 0). So we have the two paths (G1, G3, G5, G6) and (G1, G3, G5, G7) which correspond to the test templates T3 and T4 respectively*.

* Arranging the test templates into TTH is very useful in obtaining test templates with a particular property when we have huge proof trees with lots of conditions.

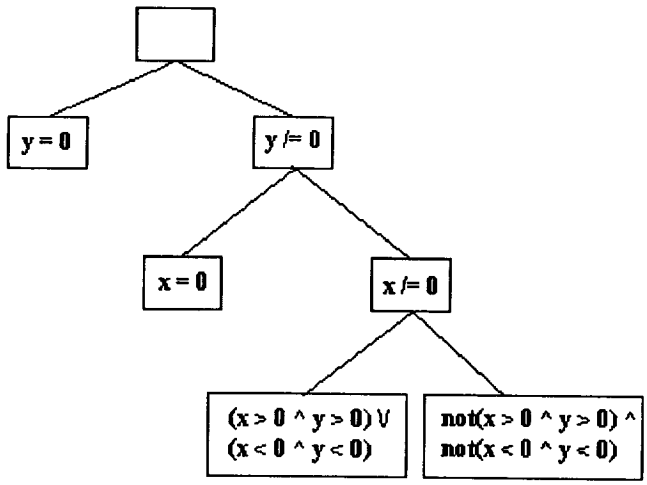


Figure 6: TTH for conj1 of Division_Result

5. A Detailed Example

In this section we will demostrate the heuristic approach to generating test templates with the 'triangle' example.

5.1 Generating Test Templates for Invalid Properties

We will first consider the case where we fail to prove the property, i.e. we will have one or more proof goals that could not be proved to be TRUE.

In this case, if we have a sequent of the form

(A1 ^ A2 ^ A3 ^ ...) --> (B1 V B2 V B3 V ...)

the test template would be (refer section 4.2)

(A1 ^ A2 ^ A3 ^ ...) ^ NOT(B1 V B2 V B3 V ...)

When we attempted to prove the property *"It is never the case that the triangle is not Equilateral or Isosceles or Scalene"* based on the triangle specification depicted in Figure 1, we failed to prove the following proof goals:

proof goal 1: Given x, y, and z are integers and y = z, prove that x = z.

proof goal 2: Given x, y, and z are integers and x = z, prove that x = y.

The actual sequents corresponding to these proof goals are

Sequent 1:

[-1] integer_pred(x!1)

[-2] integer_pred(z!1)

[-3] y!1 = z!1

|-------

[1] x!1 = z!1

Sequent 2:

[-1] integer_pred(x!1)

[-2] integer_pred(y!1)

[-3] (z!1 = x!1)

|-------

[1] x!1 = y!1

`x!1`, `y!1`, and `z!1` are the skolemized variables for `x`, `y`, and `z` respectively. `integer_pred(x!1)` means that `x!1` is an integer predicate. Based on the strategy described in section 4.2, we can derive the following test templates.

T1: integer(x) ^ integer(z) ^ (y = z) ^ not(x = z)

T2: integer(x) ^ integer(y) ^ (z = x) ^ not(x = y)

Note that the Valid Input Space (VIS) consists of all and only those (x, y, z) where x, y, and z are integers and (x, y, z) must form a valid triangle (specified in the requirements). So we have to impose these additional conditions on T1 and T2. Now, the templates will be

T1: integer(x) ^ integer(y) ^ integer(z) ^ Form_Triangle(x, y, z) ^ (y = z) ^ not(x = z)

T2: integer(x) ^ integer(y) ^ integer(z) ^ Form_Triangle(x, y, z) ^ (z = x) ^ not(x = y)

These two templates correspond to the cases where the model fails to exhibit the property ***"It is never the case that the triangle is not equilateral or isosceles or scalene".***

5.2 Generating Test Templates for Valid Properties

We demonstrate test template generation in the case of valid proofs with the following slightly modified requirements specification for the same triangle problem.

> *"Write a program that reads 3 integer values per line from input. The 3 values are interpreted as representing the lengths of the sides of the triangle. The program prints a message that states whether a triangle is Scalene, Isosceles, or Equilateral."*

Now, the program has to deal with invalid input. When we model this problem, we have to decide on the level of abstraction. For example, we can model this problem in two ways. In

the first method we can write two separate functions; one to validate the input, i.e. to ensure that input contains three integers and another function to check whether a given set of three integers form a triangle. In this approach we don't have to actually parse the input string to get the integers, we just need to make sure that the input contains three integers. In the second method we actually parse the input to get three integers and then check whether they form a triangle. We took the second approach because it will be a better way of demonstrating test template generation. The PVS specification is given in Appendix B.

5.2.1 Assumptions

As PVS does not have any in-built functions to handle strings or character arrays, we decided to work with arrays of ASCII codes instead of arrays of characters. The requirements specification does not have enough implementation details, for example it does not specify what the program output should be if the input does not have three integers. So the implementation has to make some reasonable assumptions. The model given in Appendix B works under the following assumptions

1) the numbers in the input line are separated by one or more spaces.

2) each number can be optionally preceded (immediately) by a '+' or '-' sign.

3) any leading blank spaces in the input line will be ignored.

4) every input line must be terminated by the NL character.

5) any character other than ('0'-'9', ' ', '+' or '-') shall result in the error 'ERR_INV_ARG'. If a '+' or '-' sign appears it must be immediately followed by a number (consisting of one or more digits).

6) If the input line is valid (i.e. satisfies condition 5) but has greater than three numbers then the error 'ERR_MORE_ARGS' shall be returned.

7) If the input line is valid (i.e. satisfies condition 5) but has less than three numbers then the error 'ERR_FEW_ARGS' shall be returned.

8) If the input line is valid (i.e. satisfies condition 5) and has exactly three numbers and if the three numbers do not form a triangle then the error 'ERR_NOT_A_TRIANGLE' shall be returned. Three integers form a triangle if the sum of any two integers is greater than the third integer.

9) If the input line is valid (i.e. satisfies condition 5) and has exactly three numbers and if the three numbers form an equilateral triangle then the program shall return 'Equilateral'.

10) If the input line is valid (i.e. satisfies condition 5) and has exactly three numbers and if the three numbers form an isosceles triangle then the program shall return 'Isosceles'.

11) If the input line is valid (i.e. satisfies condition 5) and has exactly three numbers and if the three numbers form a scalene triangle then the program shall return 'Scalene'.

12) If the input line is valid (i.e. satisfies condition 5) and has exactly three numbers and if the three numbers form a triangle but the triangle is not equilateral or isosceles or

scalene, then there is an error in our model (since any triangle will be one of the 3 types). In such a case the error 'Error' shall be returned.

The model (given in Appendix B) has two main functions 'Parse_Input' and 'Triangle'. 'Parse_Input' is a recursive function that takes the array of ASCII codes as input and parses it to integers. The other parameters to this function are the current index with in the array (since it is a recursive function), the number of integers found till the current index, whether a sign is found, whether the previous character is a digit or not, an array of the integers found till the current index, sign, and 'inputlen'. If the input line has three integers then the function 'Triangle' is called with those three values; otherwise an appropriate error message is returned. 'Triangle' function checks whether the three integers form a triangle, if so returns the type of the triangle. The constant 'MAXLEN' is the maximum input length and the input array indices have values in the range (0, MAXLEN - 1). We call the function 'Parse_Input' with the input (array of ASCII codes) and the index 0.

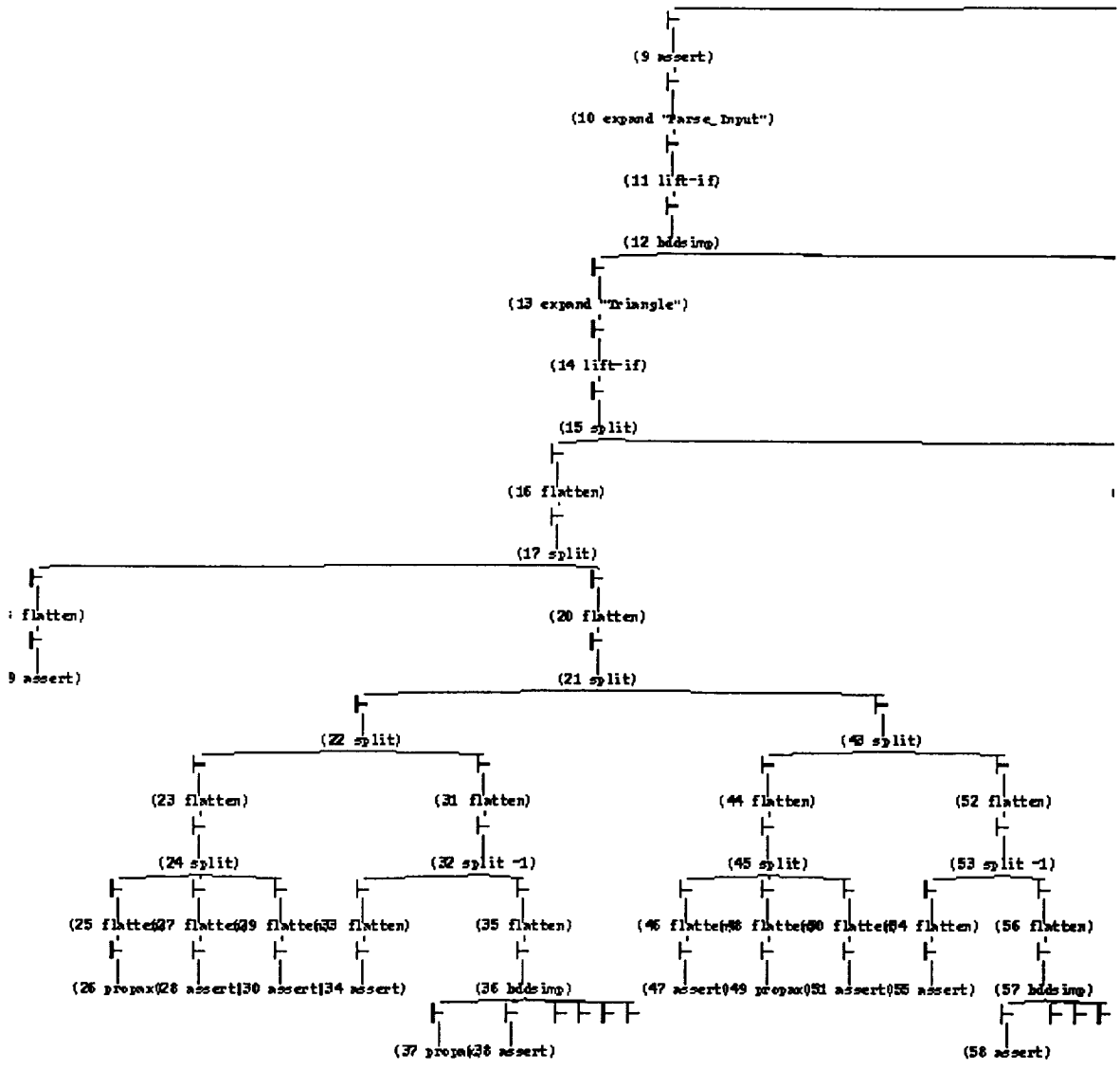


Figure 7: Partial proof tree for conj6 of modified triangle example

As we mentioned earlier, if the model returns 'Error' then there is an error (refer to assumption 12) in our model. Lets try to prove the property ***"The model will not return 'Error' for any input"***. Observe that 'Error' is returned by the function 'Triangle' when the three integers form a triangle but the triangle is not equilateral or isosceles or scalene. Also note that the 'Triangle' function is called by the function 'Parse_Input' only when the index equals input length and the input has exactly three integers. So, we need to prove only for the case when index equals inputlen-1 (since the last character must be a NL character).* So, we need to prove that ***"The model will not return 'Error' when we invoke 'Parse_Input' with the input (array of ASCII codes) and with index = inputlen - 1"***.

5.2.2 Generating Test Templates

In PVS, a conjecture can be proved in a number of different ways. For example, the above property can be proved with the single strategy

(REPEAT* (THEN* (EXPAND "Parse_Input") (GRIND))).

To generate the test templates we need a detailed proof, i.e. we need to prove the conjecture using only the very fundamental rules (such as 'flatten', and 'split') and we do not want to use strategies. We proved the above property and the proof tree is given in

* Proving the property "***The model will not return 'Error' for any input***" is cumbersome, so we simplified it.

Figure 7*. Proof tree does not show the sequents but it shows only the rules applied at each proof goal. The detailed proof with all the sequents is too lengthy to present here†. Here we will show only some selected seuqents to demonstrate the concepts.

Consider the sequent 19 in Figure 7:

{-1} edges!1(0) = edges!1(1)

{-2} edges!1(1) = edges!1(n!1) * sign!1

{-3} Error?(Equilateral)

[-4] ((edges!1(0) + edges!1(1)) > edges!1(n!1) * sign!1)

[-5] ((edges!1(1) + edges!1(n!1) * sign!1) > edges!1(0))

[-6] (edges!1(0) + edges!1(n!1) * sign!1 > edges!1(1))

[-7] inputlen!1 >= 0

[-8] integer_pred(n!1)

[-9] integer_pred(sign!1)

[-10] valid!1

[-11] a!1(inputlen!1 - 1) = 32

[-12] (1 + n!1 = 3)

|-------

---

* In Figure 7 we did not present the complete proof tree. The complete proof tree could be found at the URL: http://www.cs.wvu.edu/~pkancher/thesis/fig7.ps

† The complete proof is available at the URL: http://www.cs.wvu.edu/~pkancher/thesis/proof6.6.txt

`edges(0)`, `edges(1)` and `edges(2)` are the three integers that represent the three sides of the triangle. Note that the last antecedent [-12] indicates that `(1 + n = 3)` or `(n = 2)`. So in this sequent, `edges(n)` refers to `edges(2)`. The reason why we are multiplying `edges(2)` with '`sign`' in all the antecedents is that in our model, we allowed the numbers to be preceded by an optional sign. After we parse the number we are multiplying it with +1 or –1 depending on which sign preceded that number. For convenience, we will refer to `edges(0)`, `edges(1)` and '`edges(2) * sign`' as `e0`, `e1` and `e2` respectively.

The first two antecedents {-1} and {-2}

{-1} edges!1(0) = edges!1(1)

{-2} edges!1(1) = edges!1(n!1) * sign!1

mean that `e0 = e1` and `e1 = e2`. The third antecedent {-3}

{-3} Error?(Equilateral)

means that the symbol 'Error' is same as the symbol 'Equilateral', which is FALSE.

The next three antecedents

[-4] ((edges!1(0) + edges!1(1)) > edges!1(n!1) * sign!1)

[-5] ((edges!1(1) + edges!1(n!1) * sign!1) > edges!1(0))

[-6] (edges!1(0) + edges!1(n!1) * sign!1 > edges!1(1))

mean that (e0 + e1 > e2), (e1 + e2 > e0) and (e0 + e2 > e1). The next three antecedents do not really contribute anything to the test template and can be ignored. The last three antecedents

[-10] valid!1

[-11] a!1(inputlen!1 - 1) = 32

[-12] (1 + n!1 = 3)

mean that valid is TRUE, `a(inputlen - 1) = ' '` and `n = 2`.

Now let's construct the input string based on these conditions. We know that the `a(inputlen) = '\n'` or NL (refer to assumption 4). Since 'valid' is TRUE, we are currently parsing a number. In other words, the character preceding the current character, which is '`inputlen - 1`' is a digit. As we can see from the specification of the conjecture, we are interested only in the '`inputlen - 1`' character of the input string. So we got specific conditions that clearly specify the '`inputlen - 1`' character. As we don't have any information on the sign preceding the numbers, we will assume that the numbers can be optionionally preceded by the sign (either '+' or '-'). 'n = 2' indicates that there were two integers preceding the number that is currently being parsed. So the input string would of the form:

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

where

B : a blank space

{B} : zero or more blank spaces

[s] : optional sign (i.e. either '+' or '–' character)

D : a digit [0 – 9]

{D} : zero or more digits

'\n' : newline character

The three D{D} substrings of the above input string represent the three numbers.

A sample input string of this format will look like

" 23 +82376 –7635 \n".

Note that we generated this input string based on just the last three antecedents of the sequent. We already mentioned that the three antecedents [-7], [-8] and [-9] are trivial and hence they can be ignored while generating the test template. The conjunction of the antecedents from [-1] to [-6] ignoring {-3}, since it is FALSE (refer to heuristic 2), give

(e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

So the complete test template corresponding to this sequent would be:

T1: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

which refers to the case of "Equilateral Triangle".

As another example consider the sequent at node 24 (shown in Figure 7):

{-1} ((edges!1(0) = edges!1(1)) AND (edges!1(1) /= edges!1(n!1) * sign!1))

[-2] Error?(Isosceles)

[-3] ((edges!1(0) + edges!1(1)) > edges!1(n!1) * sign!1)

[-4] ((edges!1(1) + edges!1(n!1) * sign!1) > edges!1(0))

[-5] (edges!1(0) + edges!1(n!1) * sign!1 > edges!1(1))

[-6] inputlen!1 >= 0

[-7] integer_pred(n!1)

[-8] integer_pred(sign!1)

[-9] valid!1

[-10] a!1(inputlen!1 - 1) = 32

[-11] (1 + n!1 = 3)

|-------

[1] edges!1(0) = edges!1(1) AND edges!1(1) = edges!1(n!1) * sign!1

As in the previous case, the antecedent [-2] is FALSE and the antecedents [-6] to [-8] are trivial and can be ignored while generating the test template. The antecedents [-9] to [-11] give the same input string as in the previous case. Note that the only consequent [1], has

two conditions joined by conjunction. Only the first condition appears in the antecedent [-1]. So the consequent [1] should be negated and added to the test template. The complete test template for this sequent would now be

T2: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

i.e.,

T2: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

((e0 /= e1) V (e1 /= e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

The condition ((e0 /= e1) V (e1 /= e2)) will be TRUE if one or both the terms are TRUE. Hence we have the following three possibilities:

(e0 /= e1) is TRUE and (e1 /= e2) is FALSE.

(e0 /= e1) is FALSE and (e1 /= e2) is TRUE.

(e0 /= e1) is TRUE and (e1 /= e2) is TRUE.

Corresponding to these three cases, we will have the following three templates

T2.a: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

(e0 /= e1) ^ not(e1 /= e2) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T2.b: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

not(e0 /= e1) ^ (e1 /= e2) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T2.c: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

(e0 /= e1) ^ (e1 /= e2) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

In T2.a and T2.c, we have two terms `(e0 = e1)` and `(e0 /= e1)` joined by a conjunction and hence these templates cannot have any instances. So we can ignore T2.a, T2.c. Now, rewriting T2.b we have

T2: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

Which corresponds to one of the cases of "Isosceles triangle".

In the same way, test templates generated for the sequents 25, 26, 30, 32, 34, 35, 36 (refer to Figure 7) would be respectively T3, T4, T5, T6, T7, T8, and T9 shown below:

T3: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T4: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T5: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T6: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

Observe that the above template T6 does not have any instances.

T7: not((e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

which is equivalent to

not(e0 + e1 > e2) V not(e1 + e2 > e0) V not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

The condition not(e0 + e1 > e2) V not(e1 + e2 > e0) V `not(e0 + e2 > e1)` will be TRUE if atleast one of the terms is TRUE. Hence we have seven possibilities. The test templates corresponding to these seven possibilities would be

T7.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.e: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.g: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T8: (1 + n < 3) ^ not(1 + n = 3) ^ valid ^ a(inputlen – 1) = ' '

Which means that the input has less than 3 numbers. Hence the input string would be

IS = "{B}[[s]D{D}B{B}][s]D{D}B\n"

So the test template would be just the input string, i.e.

T8: IS = "{B}[[s]D{D}B{B}][s]D{D}B\n"

The reason why we do not have any conditions involving `e0`, `e1`, and `e2` in this case is that the input string does not have three numbers. Since the input is invalid, the function 'Triangle' won't be invoked.

T9: not(1 + n = 3) ^ not(1 + n < 3) ^ valid ^ a(inputlen – 1) = ' '

Which means that the input has more than 3 numbers. So the input string would be

IS = "{B}[s]D{D}B{B}[s]D{D} B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D}B\n"

So the test template would be

T9: IS = "{B}[s]D{D}B{B}[s]D{D} B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D}B\n"

5.2.3 Finding Errors in Specifications

As we shall see, the process of generating test templates could find bugs in the model. Consider the sequent at node 51 of Figure 7

{-1} edges!1(0) = edges!1(1)

{-2} edges!1(1) = edges!1(2)

{-3} Error?(Equilateral)

[-4] ((edges!1(0) + edges!1(1)) > edges!1(2))

[-5] ((edges!1(1) + edges!1(2)) > edges!1(0))

[-6] (edges!1(0) + edges!1(2) > edges!1(1))

[-7] inputlen!1 >= 0

[-8] integer_pred(n!1)

[-9] integer_pred(sign!1)

[-10] valid!1

[-11] a!1(inputlen!1 - 1) > 47

[-12] a!1(inputlen!1 - 1) < 58

[-13] (n!1 = 3)

|-------

[1] a!1(inputlen!1 - 1) = 32

From the above antecedents we have

`(valid = TRUE) ^ a(inputlen - 1) = [0 - 9] ^ (n = 3)`. So the input string would be of the form

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}[s]DD{D}\n"

Notice that in the earlier sequents, for e2 we had '`edges!1(n!1) * sign!1`' in all the antecedents but now we have `edges!1(2)`. Also observe that in all the earlier sequents we had `(1 + n!1 = 3)` but now we have `(n!1 = 3)`. To find out why, we can check the previous sequents of this branch of proof tree till we find the sequent with `edges(n!1)`, (which is sequent 41).

```
[-1]  inputlen!1 >= 0
[-2]  integer_pred(n!1)
[-3]  integer_pred(sign!1)
[-4]  valid!1
{-5}  a!1(inputlen!1 - 1) > 47
{-6}  a!1(inputlen!1 - 1) < 58
{-7}  Error?(Parse_Input(a!1, inputlen!1, n!1, found_sign!1, FALSE,
            edges!1 WITH [(n!1) := a!1(inputlen!1 - 1) + 10 * edges!1(n!1) - 48],
sign!1, inputlen!1))
      |-------
[1]   a!1(inputlen!1 - 1) = 32
```

As we can see in the antecedent {-7}, unlike the earlier sequents, edges!1(n!1) was not multiplied with sign!1. To find out why, we need to look at the model.

Notice that the input string has four numbers. In our model, under these conditions, we will be executing the following statement

```
Parse_Input(a, index+1, n, found_sign, false, Edges WITH [(n) :=
(edges(n)*10)+get_digit(a(index))], sign, inputlen)
```

Now, we realize the problem. When we don't have a space after the last number, in that case we are not multiplying the number with the sign as we did in the case where the last number was followed by atleast one space. Also note that we did not even increment 'n'. So instead of returning the error 'ERR_MORE_ARGS' (since the input string has four

numbers), we are ignoring the fourth number in the input string and treating it as a valid input. We found an error in our model.

Note that this actually does not violate the requirements specified for this problem. Because the problem statement does not mention what the program output should be when there are more than three numbers in the input string. But in section 5.2.1, we made some assumptions in order to meaningfully define the output of the program. The assumption 6 states that the program should return 'ERR_MORE_ARGS' if the input string has more than three numbers. Although we did not violate the requirements stated for this program, we did not do what we wanted to do (i.e., we violated assumption 6). This would have been an error if the requirements for this program had clearly specified the desired output for invalid input strings.

Also observe that we were able to prove the conjecture even though the model had an error. The reason being that the conjecture we proved verifies whether the model exhibits a specific property and that property does not have to do anything with this error. If we had tried to prove some conjecture that is some way related to this error, we would have found this error. For instance, if we try to prove the property "If the input has more than 3 integers then the program returns ERR_MORE_ARGS", then we would have found the above error.

As another example, consider the sequent at node 80 of Figure 7.

[-1] inputlen!1 >= 0

[-2] integer_pred(n!1)

[-3] integer_pred(sign!1)

[-4] a!1(inputlen!1 - 1) = 43

{-5} (n!1 = 3)

{-6} ((edges!1(0) + edges!1(1)) > edges!1(2))

{-7} ((edges!1(1) + edges!1(2)) > edges!1(0))

{-8} (edges!1(0) + edges!1(2) > edges!1(1))

{-9} (edges!1(2) = edges!1(0))

|-------

[1] valid!1

[2] 43 = 32

{3} found_sign!1

{4} edges!1(0) = edges!1(1)

{5} edges!1(1) = edges!1(2)

{6} (edges!1(0) = edges!1(2))

From the formulas [-4], {-5}, [1], and {3}, we have the following conditions*.

`a(inputlen - 1) = '+' ^ (n = 3) ^ not(valid) ^ not(found_sign)`. So the input string would be of the form

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}D{D}B{B}+\n"

We would expect the model to return 'ERR_INV_ARG' since the assumption 5 in section 5.2.1 states that "If a '+' or '-' sign appears, it must be immediately followed by a number (consisting of one or more digits)". Once again this does not violate the requirements specified for this problem but still we consider it as an error since it violates our assumption based on which we developed the model.

Similarly, the sequent 85 of Figure 7 corresponds to the input string of the form

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}D{D}B{B}-\n"

Which will also lead to an error.

As a final example, consider the sequent at node 116 of Figure 7.

[-1] inputlen!1 >= 0

[-2] integer_pred(n!1)

[-3] integer_pred(sign!1)

[-4] a!1(inputlen!1 - 1) > 47

[-5] a!1(inputlen!1 - 1) < 58

{-6} Error?(ERR_MORE_ARGS)

|-------

[1] valid!1

[2] a!1(inputlen!1 - 1) = 32

[3] a!1(inputlen!1 - 1) = 43

---

* Note that when 'valid' is FALSE, we must have already incremented 'n' and multiplied 'edges!1(n!1)' with 'sign!1' in the previous iteration of the recursion.

[4] a!1(inputlen!1 - 1) = 45

{5} (n!1 = 2)

As we mentioned earlier, we are trying to prove that the model will never return the symbol 'Error' for any input. So PVS will check all possible execution paths through the program and compare the output of the model with the symbol 'Error'. In this case, the execution path whose output would be the symbol 'ERR_MORE_ARGS' was under consideration.

From the formulas [-4], [-5], and [1] to {5}, we have the following set of conditions

a(inputlen – 1) = [0 – 9] ^ not(valid) ^ not(n = 2).

If `(n > 2)` then the output 'ERR_MORE_ARGS' is what we expect since there will be more than three numbers in the input string. In the case `(n < 2)`, we would expect the program to return 'ERR_FEW_ARGS' not 'ERR_MORE_ARGS'. So this is another error in our model. The corrected model, and the test templates are presented in Appendix C.

6. Discussion

In the earlier sections we explained the methodology of test template generation based on the properties stated in the requirements. The properties that should be exhibited by the software can be categorized into 3 classes:

1) Safety property: no execution path should exhibit this property
2) Liveliness property: some execution paths in the model should exhibit this property
3) Invariant property: all execution paths must exhibit this property.

As an example, consider the requirements for the mutual exclusion problem in multi-programming environment [7].

1) Only one process can execute its critical section at any one time.
2) When no process is executing in its critical section, any process that requests entry to its critical section must be permitted to enter with out delay.
3) When two or more processes compete to enter their respective critical sections, the selection cannot be postponed indefinitely.
4) No process can prevent any other process from entering its critical section indefinitely; that is every process should be given a fair chance to access the shared resource.

The safety property is the logical negation of the invariant property. So any safety property can be paraphrased to get an equivalent invariant property. For example, the property 1 in the above specification can be thought of as a safety property "There is no execution path in which more than one process is in its critical section simultaneously". It can be paraphrased as an invariant property "In every execution path, there is atmost one process in its critical section at any given instant". We can directly specify these properties in PVS as conjectures and test templates can be generated as explained in the earlier sections.

In the remainder of this section we'll demonstrate the significance of test template generation from the proof tree with a practical example. This example will also illustrate the significance of testing for safety properties in real world applications. The requirements specification for the example is given below:

```
Procedure: Scheduler (called every 125ms)
        Check_for_Overrun;
        Run_Tasks;
End Scheduler;

Procedure: Check_For_Overrun
        For I in 1..Task_List.num {
                If (Process_State(I) == RUNNING){
                        Set_Error_Register(); -- task did not finish in the earlier run.
                        Halt_System();
                }
                Else
                        Process_State(I) := WAITING; -- ready to be executed in this run.
        }
End Check_For_Overrun;

Procedure: Run_Tasks
        Current_Task_Index := 1;
        While(Current_Task_Index <= Task_List.num){
                Process_State(Current_Task_Index) := RUNNING;
                Execute_Task(Current_Task_Index); -- jump to new address and start executing the task
                Process_State(Current_Task_Index) := COMPLETE;
                Current_Task_Index++;
        }
        Wait_For_Interrupt(); -- sleep till the 120ms hardware interrupt occurs.
End Run_Tasks;
```

Figure 8: Pseudo code for the scheduler

Write a program for a scheduler that will schedule a set of fixed number of tasks in a fixed order. Initially all the tasks will be in the "WAITING" state. As soon as a task is scheduled, its state is changed to "RUNNING". Every task has finite execution time (not a constant, since it depends on a number of factors). A hardware interrupt will be generated every 120ms and will halt the scheduling process immediately. It is intended that all the tasks should be scheduled and finish execution within this time limit. After 5ms of dead time (during this period the bus will be inactive), the

> *scheduler is re-started and it should start executing the tasks from the first one. If all the tasks are not finished in the time limit then the scheduler should detect this when it is entered again after the 5ms dead time and set the 0th bit in the Error register (#6A5F) and halt the system.*

Note that testing an implementation of the scheduler for functional correctness is a difficult problem because there are an infinite number of states where an interrupt can occur. Testing a program for all these cases is not only impractical but is not possible. So we cannot prove the functional correctness of this program by conventional testing methods. This scheduler might be part of a safety critical system and hence proof of its functional correctness may be essential. We might model the scheduler algorithm in a formal language and prove its functional correctness. However, our formal specification itself might not be an exact representation of the actual implementation. There might exist some inconsistencies between the model and the actual implementation. So, If we find some error in the specification, we would want to test the implementation for that error. If the formal specification fails to exhibit a property, we would like to find a test case corresponding to this failure and test our implementation for that particular test case.

Modeling interrupts in PVS is difficult, so here we present only the pseudo code. The pseudo code for scheduler is shown in Figure 8. The scheduler is called every 125ms after the 5ms dead interval. Scheduler first calls Check_for_Overrun to see whether any of the tasks are in the RUNNING state when the 120ms hardware interrupt occurred. If so, it'll set the error register and halt the system. Otherwise all the processes' states are set to

WAITING. The scheduler then calls the function Run_Tasks to execute the tasks. Run_Tasks will set each task' state to RUNNING just before jumping to the tasks' address. After the task is finished, control is returned to the Run_Task procedure and the tasks' state is set to COMPLETE. This procedure is repeated for all the tasks starting with the first.

Now, we are interested in knowing whether this code will satisfy the requirement "If any of the tasks did not finish in the previous run, then the 0^{th} bit of the Error Register should be set and the system should be halted". If we model this scheduler algorithm in PVS, we can specifiy the above property as "Error Register is set iff there exists a task that did not finish in the earlier run". When we attempt to prove the above conjecture in PVS, we will fail to prove it. The model (assuming it is developed based on the pseudo code presented in Figure 8) will not be able to detect that some tasks did not execute if the hardware interrupt occurs after execution of one task and before the start of execution of the next task (i.e. when the control is in scheduler). This might lead lead to stack overflow and other serious problems. Finding such a subtle problem with conventional testing would be very difficult if not impossible.

7. Conclusions

Conventional testing methods fail to prove the correctness of the program because of very large input space. In the presence of evolving specifications and code changes it is not sufficient to prove the correctness of the specification because of the inconsistencies that exist between the formal specification and implementation. We suggested and demonstrated a new method of testing software based on the formal specification. In this approach we will be generating test templates corresponding to the properties stated in the requirements. A brief overview of the procedure for generating test templates follows.

The program that needs to be verified is modeled in the fomal specification/verification system, Prototype Verification System (PVS). The properties that should be exhibited by the software are stated as conjectures in the model. PVS proof checker consists of a number of proof commands that can be used to prove the conjectures. The proof commands applied in order to prove the conjecture can be built into a proof tree. Based on the proof tree we generate test templates corresponding to the conjecture (or the property).

In the case we fail to prove a conjecture, we will generate test templates corresponding to the proof goals that we could not prove. Inability to prove a conjecture based on the model does not necessarily mean that the actual implementation has some error. It could be

because of the inconsistencies between the model and the implementation or because of the insufficient information in the model (i.e., in the case of partial specification / verification). So we generate test templates corresponding to the cases for which the model fails to exhibit the specified property. Then, we can test the actual implementation for these cases to see whether the actual implementation also has the error that has been identified in the model. If so, we correct both the implementation and the model and try to prove all the properties based on the corrected model. Otherwise, we correct the model to rectify the problem.

If we are successful in proving a conjecture, then we derive test templates corresponding to each of the leaves of the proof tree. However, problems arise in devising a set of rules for generating test cases, since there is no unique way to prove a property. Different set of proof commands (or proof strategies) can be used to prove the same conjecture in different ways. The proof tree and hence the generated test templates vary according to the set of proof commands used to prove the property. So we proposed a set of heuristics that aid in generating test templates for valid properties.

We also claimed that proving a conjecture successfully does not imply that the implementation or the model is correct. It only means that the stated conjecture is true with respect to the model. In the case where the requirements specification of a problem does not have enough details, the implementation has to make certain reasonable assumptions. To this end, we demonstrated that errors (with respect to the assumptions made) might

exist in the model (or in the implementation) even though we could prove the conjectures. We exemplified the approach to finding these errors during the process of test template generation.

We also presented a strategy for organizing the test templates into the Test Template Hierarchy (TTH), which will be useful in identifying test templates with a specific property. TTH is especially useful in the case of complex problems where the number of test templates generated could easily become unmanageable. Finally, we discussed the significance of our approach to testing software with a practical example.

The most interesting area for future work would be the development of a tool based on our approach. The tool should automatically parse the generated proof for every conjecture and shall derive the test templates corresponding to them. The process of organizing the test templates into a TTH can also be automated. The tool should also allow the user to select test templates with a specific property. The user can then test the actual implementation for a set of selected test cases based on the test templates.

It will also be interesting to study how each of the proof commands provided by the PVS proof checker affect the proof tree. This will help improving the set of heuristics provided in section 4.3. It will also form the basis for automating the test template generation and building the tool described earlier.

References

[1] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structrured Programming, APIC Studies in Data Processing, no. 8 Academic Press, 1972.

[2] Phil Stocks and David Carrington, "A Framework For Specification-Based Testing", IEEE Trans. on Software Eng., vol. 22, no. 11, pp. 777-793, Nov. 1996.

[3] Richard L. Ford and Lawrence M. Smith, "Specification-Based Event-Trace Testing", http://www.opengroup.org/wwww/formal-methods.

[4] Merlin Hughes and David Stotts, "Daistish: Systematic Algebraic Testing for OO Programs in the Presence of Side-effects", ISSTA '96, pp. 53-61.

[5] Bruno Dutertre and Victoria Stravridou, "Formal Requirements Analysis of an Avionics Control System", IEEE Trans. on Software Eng., vol. 23, no. 5, pp. 267-278, May 1997.

[6] Glenford J. Myers, "The Art of Software Testing", John Wiley & Sons, 1979.

[7] Mukesh Singhal and Niranjan G. Shivaratri, Advanced Operating Systems, McGraw-Hill, 1994.

[8] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandyam Srivas, "A Tutorial Introduction to PVS", http://www.csl.sri.com/pvs/examples/wift-tutorial/wift-tutorial.html.

[9] Ricky Butler, "An Elementary Tutorial on Formal Specification and Verification Using PVS", http://atb-www.larc.nasa.gov/ftp/larc/PVS-tutorial/.

[10]John Rushby and David W.J. Stringer-Calvert, "A Less Elementary Tutorial for the PVS Specification and Verification System", http://www.csl.sri.com/pvs/examples/elementary-tutorial/csl-95-10.html.

[11]Sam Owre and John Rushby, "FME '96 Tutorial: An Introduction to Some Advanced Capabilities of PVS", http://www.csl.sri.com/pvs/examples/fme96/fme96-tutorial.html

[12]Sam Owre, "Overview of the PVS Verification System", http://www.csl.sri.com/pvs/overview.html

[13] N. Shankar, S. Owre and J. M. Rushby, "The PVS Proof Checker: A reference manual", http://www.csl.sri.com/reports/postscript/pvs-prover.ps.gz

Appendix A: PVS Primer

Prototype Verification System (PVS) is a specification and verification system developed by SRI International. It consists of a specification language integrated with support tools and a theorem prover. PVS has a sophisticated type system containing predicate subtypes, dependent subtypes and abstract datatypes such as lists and trees. The standard PVS types include numbers (integers, reals, naturals, etc.) records, tuples, arrays, functions, sets, sequences, lists, and trees, etc. PVS has a very strong type checking system that will automatically generate proof obligations whenever there is some ambiguity. PVS specifications are organized into parametrized theories that may contain assumptions, definitions, axioms and theorems. PVS expressions provide the usual arithmetic, logical operators and quantifiers. Name overloading is allowed in PVS. An extensive prelude of built-in theories provides numerous useful definitions and lemmas.

PVS has a powerful interactive theorem prover / proof checker. The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. User defined procedures can combine the primitive inferences to yield higher-level proof strategies. Proofs yield scripts that can be edited, attached to additional formulas, and

rerun. This allows many similar theorems to be proved efficiently. The application of a procedure can either generate further subgoals or prove a subgoal. PVS's automation suffices to prove many straightforward results automatically.

Numerous tutorials, documents and research papers are available on PVS. For more information on PVS please refer to [8]-[13].

Appendix B: Modified Triangle Problem Specification

```
triangle6 : THEORY
BEGIN
        x, y, z: VAR int
        char_type: TYPE = {x:nat | x < 256}
        Return_type: TYPE = {ERR_FEW_ARGS, ERR_MORE_ARGS, ERR_INV_ARG,
        ERR_NOT_A_TRIANGLE, Scalene, Isosceles, Equilateral, Error}
        MAXLEN : posnat;
        Index_type: TYPE = {n:nat| n < MAXLEN}
        Character_Array_type: TYPE = ARRAY [Index_type --> char_type]
        Integer_Array_type: TYPE = ARRAY [Index_type --> int]
        valid, found_sign : VAR boolean
        a : VAR Character_Array_type
        edges : VAR Integer_Array_type
        v,n,sign : VAR int
        inputlen : VAR nat
        current, index: VAR Index_type

        NullCharArray(i: Index_type): char_type = 10
        NullIntArray(i: Index_type): int = 0


        isdigit?(c: char_type): boolean = IF c > 47 AND c < 58 THEN True ELSE False ENDIF
        get_digit(c: char_type): int = c - 48
        issign?(c: char_type): boolean = IF c = 43 OR c = 45 THEN True ELSE False ENDIF
        get_sign(c: char_type): int = IF c = 43 THEN 1 ELSIF c = 45 THEN -1 ELSE 0 ENDIF
        isspace?(c: char_type): boolean = IF c = 32 THEN True ELSE False ENDIF
        isnewline?(c: char_type): boolean = IF c = 10 THEN True ELSE False ENDIF

        Triangle(x, y, z): Return_type =
                IF ((x + y) > z) AND ((y + z) > x) AND ((z + x) > y) AND (x > 0) AND (y > 0) AND (z >
                0) THEN
                        IF x = y AND y = z THEN Equilateral
                        ELSIF ((x = y) AND (y /= z)) OR
                                ((y = z) AND (x /= z)) OR
                                ((x = z) AND (x /= y)) THEN Isosceles
                        ELSIF x /= y AND y /= z AND z /= x THEN Scalene
                        ELSE Error
                        ENDIF
                ELSE    ERR_NOT_A_TRIANGLE
                ENDIF
```

```
        Parse_Input(a, index, n, found_sign, valid, edges, sign, inputlen): RECURSIVE Return_type =
                IF index = inputlen THEN
                        IF ((n = 3) AND valid = False) OR ((n = 2) AND valid = True) THEN
                                Triangle(edges(0), edges(1), edges(2))
                        ELSIF ((n < 3) AND valid = False) THEN ERR_FEW_ARGS
                        ELSE ERR_MORE_ARGS
                        ENDIF
                ELSE
                        IF valid = true THEN
                                IF isspace?(a(index)) THEN % if space then
                                        Parse_Input(a, index+1, n+1, False, false,
                                                edges WITH [(n) := edges(n)*sign], 1, inputlen
                                ELSIF isdigit?(a(index)) THEN
                                        Parse_Input(a, index+1, n, found_sign, false, edges WITH [(n) :=
                                                (edges(n)*10)+ get_digit(a(index))], sign, inputlen)
                                ELSE ERR_INV_ARG
                                ENDIF
                        ELSE
                                IF isspace?(a(index)) AND found_sign = False THEN % if space then
                                        Parse_Input(a, index+1, n, False, false, edges, sign, inputlen)
                                ELSIF issign?(a(index)) AND found_sign = False THEN % if +, - then
                                        Parse_Input(a, index+1, n, True, false, edges,
                        get_sign(a(index)), inputlen)
                                ELSIF isdigit?(a(index)) THEN
                                        Parse_Input(a, index+1, n, found_sign, true, edges WITH [(n) :=
                                        (edges(n)*10)+get_digit(a(index))], sign, inputlen)
                                        ELSE ERR_INV_ARG
                                ENDIF
                        ENDIF
                ENDIF

        MEASURE (LAMBDA a, index, n, found_sign, valid, edges, sign, inputlen: inputlen - index);


        %proved;
        %Parse_Input conjecture6; i/p string GENERIC! case 'index = inputlen - 1"
        parse_conj6: CONJECTURE Parse_Input(a, inputlen - 1, n, found_sign, valid,
                        edges, sign, inputlen) /= Error
END triangle6
```

Appendix C: Complete Triangle Problem Specification

Corrected PVS specification of the modified 'triangle' example and the test templates

```
triangle8 : THEORY
BEGIN
        x, y, z: VAR int
        char_type: TYPE = {x:nat | x < 256}
        %Triangle_type: TYPE = {ERR_NOT_A_TRIANGLE, Scalene, Isosceles, Equilateral, Error}
        %Validate_error_type: TYPE = {ERR_FEW_ARGS, ERR_MORE_ARGS, ERR_INV_ARG,
        CORRECT}
        Return_type: TYPE = {ERR_FEW_ARGS, ERR_MORE_ARGS, ERR_INV_ARG,
                                ERR_NOT_A_TRIANGLE, Scalene, Isosceles, Equilateral, Error}
        MAXLEN : posnat;
        Index_type: TYPE = {n:natl n < MAXLEN}
        Character_Array_type: TYPE = ARRAY [Index_type --> char_type]
        Integer_Array_type: TYPE = ARRAY [Index_type --> int]
        valid, found_sign : VAR boolean
        a : VAR Character_Array_type
        edges : VAR Integer_Array_type
        v,n,sign : VAR int
        inputlen : VAR nat
        current, index: VAR Index_type

        NullCharArray(i: Index_type): char_type = 10
        NullIntArray(i: Index_type): int = 0

        %Modeling the input.
        %% This model works under the following assumptions:
        %% 1. All the values are seperated by one or more spaces.
        %% 2. An integer can be optionally (immediately) preceded by a sign ('+' or '-')
        %% 3. Any character other than ('0'-'9', '+', '-') will result in an error.
        %% 4. The input line will be terminated by a NL character.


        %lemma1: LEMMA FORALL (x,y: char_type): char(x) = char(y) IFF x = y

        isdigit?(c: char_type): boolean = IF c > 47 AND c < 58 THEN True ELSE False ENDIF
        get_digit(c: char_type): int = c - 48
```

```
issign?(c: char_type): boolean = IF c = 43 OR c = 45 THEN True ELSE False ENDIF
get_sign(c: char_type): int = IF c = 43 THEN 1 ELSIF c = 45 THEN -1 ELSE 0 ENDIF
isspace?(c: char_type): boolean = IF c = 32 THEN True ELSE False ENDIF
isnewline?(c: char_type): boolean = IF c = 10 THEN True ELSE False ENDIF

Triangle(x, y, z): Return_type =
        IF ((x + y) > z) AND ((y + z) > x) AND ((z + x) > y) THEN
                IF x = y AND y = z THEN Equilateral
                ELSIF ((x = y) AND (y /= z)) OR
                        ((y = z) AND (x /= z)) OR
                        ((x = z) AND (x /= y)) THEN Isosceles
                ELSIF x /= y AND y /= z AND z /= x THEN Scalene
                ELSE Error
                ENDIF
        ELSE    ERR_NOT_A_TRIANGLE
        ENDIF
Parse_Input(a, index, n, found_sign, valid, edges, sign, inputlen): RECURSIVE Return_type =
        IF index = inputlen THEN %% index > MAXLEN??
                IF ((n = 3) AND valid = False) OR ((n = 2) AND valid = True) THEN
                        Triangle(edges(0), edges(1), edges(2))
                ELSIF ((n < 3) AND valid = False) OR (n < 2) THEN ERR_FEW_ARGS
                ELSE ERR_MORE_ARGS
                ENDIF
        ELSE
                IF valid = true THEN
                        IF isspace?(a(index)) THEN % if space then
                                Parse_Input(a, index+1, n+1, False, false,
                                        edges WITH [(n) := edges(n)*sign], 1, inputlen)
                        ELSIF isdigit?(a(index)) THEN
                                IF isnewline?(a(index+1)) THEN
                                        Parse_Input(a, index+1, n+1, found_sign, false, edges
                                WITH [(n)
                                                := ((edges(n)*10)+get_digit(a(index))) *
                                        sign], sign,
                                                        inputlen)
                                ELSE
                                        Parse_Input(a, index+1, n, found_sign, false, edges
                                WITH [(n) :=
                                                (edges(n)*10)+get_digit(a(index))], sign,
                                        inputlen)
                                ENDIF
                        ELSE ERR_INV_ARG
                        ENDIF
                ELSE
                        IF isspace?(a(index)) AND found_sign = False THEN % if space then
                                Parse_Input(a, index+1, n, False, false, edges, sign, inputlen)
                        ELSIF issign?(a(index)) AND found_sign = False THEN % if +, - then
                                IF isdigit?(a(index+1)) THEN
                                        Parse_Input(a, index+1, n, True, false, edges,
                                                get_sign(a(index)), inputlen)
                                ELSE ERR_INV_ARG
```

```
                                ENDIF
                            ELSIF isdigit?(a(index)) THEN
                                IF isnewline?(a(index+1)) THEN
                                    Parse_Input(a, index+1, n+1, found_sign, true,
                                        edges WITH [(n) :=
                                            ((edges(n)*10)+get_digit(a(index)))*
                                        sign],
                                            sign, inputlen)
                                ELSE
                                    Parse_Input(a, index+1, n, found_sign, true,
                                        edges WITH [(n) :=
                                    (edges(n)*10)+get_digit(a(index))],
                                            sign, inputlen)
                                ENDIF
                            ELSE ERR_INV_ARG
                            ENDIF
                    ENDIF
            ENDIF

        MEASURE (LAMBDA a, index, n, found_sign, valid, edges, sign, inputlen: inputlen - index);

        parse_conj1: CONJECTURE Parse_Input(a, inputlen - 1, n, found_sign, valid, edges, sign, inputlen)
/= Error

END triangle8
```

The proof tree for the conjecture "parse_conj1" of this model will be very similar to Figure 7. We can generate test templates from the proof tree exactly the same way we generated test templates in the section 5.2.2. Here we present the final test templates

T1: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T2: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T3: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T4: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T5: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T6: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.e: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T7.g: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B\n"

T8: IS = "{B}[[s]D{D}B{B}][s]D{D}B\n"

T9: IS = "{B}[s]D{D}B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D}B\n"

T10: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T11: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T12: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T13: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T14: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T15: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.e: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T16.g: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}\n"

T17: IS = "{B}[[s]D{D}B{B}][s]D{D}\n"

T18: IS = "{B}[s]D{D}B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D}\n"

T19: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}\n"

T20: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T21: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T22: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T23: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T24: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T25.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}\n"

T25.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T25.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T25.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T25.e: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T25.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T25.g:(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D} B{B}\n"

T26: IS = "{B}[[s]D{D}B{B}][s]D{D} B{B}\n"

T27: IS = "{B}[s]D{D}B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D} B{B}\n"

T28: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]Dn"

T29: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T30: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T31: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T32: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T33: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.e: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T34.g: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D\n"

T35: IS = "{B}[[s]D{D}B{B}][s]D\n"

T36: IS = "{B}[s]D{D}B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D\n"

T37: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T38: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T39: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T40: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T41: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T42: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.e: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T43.g: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}+\n"

T44: IS = "{B}[[s]D{D}B{B}][s]D{D}B{B}+\n"

T45: IS = "{B}[s]D{D}B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D}B{B}+\n"

T46: (e0 = e1) ^ (e1 = e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T47: (e0 = e1) ^ (e1 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T48: (e1 = e2) ^ (e0 /= e2) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T49: (e0 = e2) ^ (e0 /= e1) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T50: (e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0) ^ (e0 + e1 > e2) ^ (e1 + e2 > e0) ^

(e0 + e2 > e1) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T51: (e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1)

^ not((e0 /= e1) ^ (e1 /= e2) ^ (e2 /= e0)) ^ not((e0 = e1) ^ (e1 /= e2))

^ not((e1 = e2) ^ (e0 /= e2)) ^ not((e0 = e2) ^ (e0 /= e1))

^ not((e0 = e1) ^ (e1 = e2)) ^ IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.a: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.b: (e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.c: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.d: not(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.e:(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ not(e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.f: not(e0 + e1 > e2) ^ (e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T52.g:(e0 + e1 > e2) ^ not(e1 + e2 > e0) ^ (e0 + e2 > e1) ^

IS = "{B}[s]D{D}B{B}[s]D{D}B{B}[s]D{D}B{B}-\n"

T53: IS = "{B}[[s]D{D}B{B}][s]D{D}B{B}-\n"

T54: IS = "{B}[s]D{D}B{B}[s]D{D}{B{B}[s]D{D}}B{B}[s]D{D}B{B}-\n"