STPG
An Assistant for Generating Test Program Software

Charles E. Matthews WLO Somers, NY CEM @ RHQVM08

Abstract: Automatic program generation has been investigated within the AI field for some time. The most successful approach thus far is the Programmer's Apprentice Project at MIT. The State Test Program Generator (STPG) utilizes some of the concepts from the PA Project to generate test software. The current implementation of STPG is targeted for generating test cases to test the knowledge bases for a hardware configurator (the IBM Solution Manager project). This methodology, however, is equally applicable to other target domains. To verify this approach, an additional generator for creating test cases for hardware designs is planned.

Introduction: This paper describes the current development and use of an automated assistant for generating test case software. Test generation is an important issue for the development process because it is the only mechanism available for determining whether the product which was actually built matches the "intentions" of the product designer. When the details of these "intentions" pass a certain limit of complexity, human testers become overloaded with the intricacies of the interrelationships and dependencies between portions of the design. Other test generation tools, e.g. the RTPG tool used by the AS/400 and RS/6000 hardware design groups, rely upon a random or pseudo-random generation of test instruction sequences for their test cases. The STPG approach uses a data driven model of the design to guide the generation of test instructions. The model is dependent upon the representation of the design functions within a database or knowledge base.

$$\begin{tikzcd}
 & \text{PKB} \arrow[d] & & & \\
\text{User} \arrow[r] & \text{STPG Tool} \arrow[r] & \text{Test Case Instructions} \arrow[r] & \text{Driver/ Translator} \arrow[r] & \text{Executable Platform}
\end{tikzcd}$$

Figure 1 - STPG Process

The ISM project captures all of the product specific configuration data in a Product Knowledge Base (PKB). The PKB has a pre-defined syntax and structure which is interpreted by the STPG tool to create a model of the design. This model is used interactively by the user to create test case instructions. These instructions are interpreted by a driver for a given executable platform. This final driver step is important for portability reasons, but it is not required by the methodology.

Scope: Automatic test generation cannot solve all testing problems nor can it find every type of error. For the current environment design errors are separated into three classes.

1. The PKB contains incorrect knowledge which is semantically correct.

2. The PKB lacks knowledge.

3. The PKB contains incorrect knowledge which is semantically incorrect.

Type 1 errors appear when the design database contains information which appears logically and semantically correct, but the data is functionally incorrect. For a hardware configuration this type of error is illustrated if a disk drive is configured with the wrong adapter card. Logically, the design is correct because an adapter card is attached to the drive, however, it is the wrong adapter card. Unfortunately, these errors are extremely difficult to find with this approach because the test generation model uses a data driven paradigm. The test generator interprets the design model directly from the PKB data. If the model is semantically sound but functionally incorrect, the test software will generate test cases for the incorrect design. These errors will only be found by inspecting the test results from the viewpoint of the "correct" design model. In most cases this model only exists as a mental model in the designer's mind.

Type 2 errors illustrate the "incomplete knowledge problem". A special case of these errors are known as dangling objects. A dangling object is a PKB object which either is not referenced by anything else or does not itself reference anything else. Dangling objects can be detected programmatically so special test cases for them are rarely required.

Other type 2 errors are diagnosed with the same approach as type 1 errors. Inspection of the test case results with the designer's mental model of the product is required to find these errors.

Type 3 errors comprise the majority of the errors found by testing. These errors are the targeted objective of most of the test cases generated by the STPG program.

Methodology: Each test case has three components: an environment description, actions, and result tests. The environment description sets certain global parameters which remain constant for the duration of the test case. In the ISM application the environment consists of the configuration date, the ship date, the country code, etc. Test case actions and result tests are usually intermixed instructions. A sequence of actions will be specified. Because the STPG program has derived a model of the design, the program can infer some of the responses to the action commands. The actual responses of the design are compared to the STPG model. Anomalies are reported as test case failures to the testing team.

The ISM project has an additional testing complication in that it is itself a data-driven program. Thus, a configuration scenario must be identified to the STPG tool. This scenario is built by identifying the subsystems which will be configured for the ensuing test cases. A knowledge base for a 3090 processor may contain information about many subsystems, but a configuration scenario identifies the subset of those subsystems which will be used for the current test generation session.

STPG Menus: The initial program screen contains four pull-down menus: Subsystems, Test Abstractions, Environment Parameters, and the Quit Options. The PKBs loaded into the STPG tool determine which subsystems are present for a configuration. Selecting the Subsystem Menu allows the user to select the desired subset. The Environment Parameters menu lists the global test parameters. Selecting a parameter allows its value to be edited. The Quit menu identifies when the test case session is completed and whether or not it should be saved.

The intelligence of the STPG tool, however, is gated primarily by the sophistication of the Test Abstraction menu. Each item of this menu corresponds to a different cognitive operation which is typically done by a test case author. Whenever the STPG tool is modified to a new testing domain, a knowledge engineer must study that domain enough to identify the set of generic test abstractions which constitute the problem domain for that testing environment.

Test abstractions for the configuration domain follow in the next section. Appropriate test abstractions for a hardware test domain would be: test the data flow, test multiple transfers across the SCSI bus, test the bus arbitration logic, etc. Appropriate test abstractions for a microcode test domain would be: test virtual address translation, test data accesses across page boundaries, test message handling among the SEND/RECEIVE queues, etc. Implementing the STPG tool for a given test domain requires a knowledge acquisition effort for that domain. However, once the test abstractions for that domain are determined, these abstractions should be similar for different designs within the same domain.

Test Abstractions: The problem domain for configurator testing contains approximately twenty abstract test operations. The following list is an example subset of those operations.

- Device autoselects another device
- Device backtracking
- Device requires a feature, mandatory
- Device requires a feature, choice permitted
- Device requires a resource, mandatory
- Device requires a resource, choice permitted
- Feature requires another feature through a secondary product requirement
- Feature requires a resource provided by the device

Each abstract test operation requires one or more rules which replicate a test case author's actions for testing that operation with the given PKB model. Depending upon the specific operation, actions may proceed forward in time or backwards in time. For example, the rule for the action **Device autoselects another device** creates the following sequence of events.

Some PKB devices are automatically selected by the configuration of a previous device. Identifying a PKB device which is to be selected automatically becomes a goal whose resolution is dependent upon the previous selection/configuration of the antecedant device. Thus, the required actions are running backwards in time. If the antecedant device has prerequisite conditions upon it, the chain of configuration events can become lengthy. For example, if device DA-PS30 is automatically selected by device DASD-30 and DASD-30 requires CPU-2, then a user request to autoselect device DA-PS30 results in the following test instructions.

```
Select CPU-2;
Select DASD-30;
  Test for the presence of DA-PS30;
```

In this case the test user did not need to know which devices actually caused the automatic selection of device DA-PS30. The STPG tool created the appropriate sequence of instructions based upon the design model inferred from the PKB.

On the other hand, the rule for the action Device Backtracking tests the configurator logic for removing previously configured devices from a given configuration. Selecting the device to be removed can only be done once a test action selecting that device has already been specified. Thus, the actions are moving forward in time. Consider the test case listed above. If the user elects to remove device DASD-30 from the configuration, the test case becomes:

```
Select CPU-2;
Select DASD-30;
  Test for the presence of DA-PS30;
Remove DASD-30;
  Test for the absence of DASD-30;
  Test for the absence of DA-PS30;
```

In this example the test case author has only specified two actions, but the STPG program generated the correct test case based upon those abstract actions. This tool allows the test case author to specify test cases at a higher level of abstraction than ever allowed before.

Summary: The STPG program allows the test case author to create test cases interactively with two distinctions from previous tools. 1) The tool maintains a model of the complex dependencies between objects in the PKB. As long as this model is correct, then STPG generates functionally correct test cases at all times. 2) The tool allows the test case author to think at a higher cognitive level of abstraction. The test engineer should concentrate upon designing tests. The tool should create the instruction sequence which implements those tests.

There are a number of advantages to this approach.

1. Increased productivity: This tool allows the generation of test case software with much less time and manpower resources. Fewer people can create larger test case libraries than current testing organizations

can do today. The STPG tool will shift the limiting test bottleneck from test case generation to somewhere else.

2. More reliable testing: Because the STPG tool generates test cases based upon its model of the design, consistent test cases are generated. If the model is correct, no test case errors will ever occur. Thus, we have a tool which always generates functionally correct test software.

3. Faster response to design changes: Design changes are a fact of life. Current test environments require the test engineers to track and to recompile their private mental models of the design any time a design change occurs. The STPG tool replaces all of that with one "golden source" for the design.

A "proof of concept" version of STPG is completed. It is coded in the ART-IM shell running in MS-DOS on the PS/2. A production version of STPG for the ISM PKBs is scheduled for 9/91. The production version will be coded in KEE running on the PC/RT. Negotiations with one of the hardware development groups in Rochester is underway to implement STPG for hardware test cases.