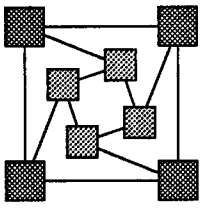
Automatic Test Data Generation Tool for Large-Scale Software Systems[1]

Final Report

A.Kolawa[2], B.Strickland, A.Hicken

ParaSoft Corporation
2031 S. Myrtle Ave.
Monrovia, CA 91016

Phone: (818) 305-0041
FAX: (818) 305-9048



Contract: DASG60-94-C-0051
May 26, 1994

---

1. Research sponsored by BMDO and managed by U.S.Army, Strategic Defense Command, Huntsville, Alabama, under the SBIR program. Technical Officer CSSD-AT-P Claire McCullough
2. Principal Investigator

ParaSoft Corporation
2031 S. Myrtle Ave., Monrovia, CA 91016 Phone (818) 305-0041 FAX (818) 305-9048

Defense Technical Information Center
P.O. Box 1500
Cameron Station
Alexandria, VA 22304-6145

December 15, 1994

RE: Contract **DASG60-94-C-0051**

Sir,

Please find enclosed our final report for SBIR contract# **DASG60-94-C-0051**, May 26, 1994 - *Automatic Test Data Generation Tool for Large-Scale Software Systems*[1]. If you would like additional copies, versions on electronic media, etc. we would be happy to supply them.

Please contact me if you have any questions or comments. I would be happy to hear from you.

Sincerely,

Arthur Hicken
ParaSoft Corporation.

1. Research sponsored by BMDO and managed by U.S.Army, Strategic Defense Command, Huntsville, Alabama, under the SBIR program. Technical Officer CSSD-AT-P Claire McCullough

Table of Contents

1. Phase I Effort Summary

In our Phase I effort we undertook a project which many people would consider impossible to execute. We proposed to build an automatic test-case generation tool and deliver a working prototype of it. Automatic test case generation is considered one of the most difficult and unsolvable problems in computer science. The task is high-risk, hard to do, and nobody in industry is even trying to solve the problem. In addition to that we set out to build a prototype of the tool on a Phase I budget. This was only possible because of the high reusability of the software and technology which we already have in-house. If we weren't able to re-use the software, the results which we present in this report would require 2-3 years effort.

The automatic test case generation problem is so hard and such high-risk that we would not attempt to solve it without the possibility of SBIR funding.

We believe that we achieved our Phase I goals. The testing tool prototype was built and it works on problems of significant size. Our innovation was in several areas. One was in applying and extending existing test-case generation algorithms. We enhanced and refined algorithms that were already available. Another was the development of new techniques used to combine and apply the aforementioned algorithms to the problem. The unique combination of algorithms and how they are applied to the problem was the key to our success.

The results of testing the prototype are encouraging and suggest that the research, if continued, can lead to real breakthroughs in the way people test their software. This in turn would lead to high payoffs. However we realize that there is still a lot of work before the prototype we have developed can be used successfully with large software systems.

Before reporting our results we would like to describe in more detail the software testing process, and the role and significance of an automatic test-case generation tool.

Software testing is very labor-intensive and expensive; it accounts for approximately 50% of the development costs of a software system. At the same time, software testing is critical in achieving quality software. Software developers know that there is no possibility of building reliable and successful software products without sound, efficient, and thorough testing. Because software testing is so important, and accounts for such a large part of the software development cost, any advances in software testing technology yield large benefits.

Recent studies [Man94] indicate that software testing presently done in the industry is not adequate. On average, a typical company tests only 30% of the code it develops. The remaining 70% is never tested before it is shipped. The reason for this is that most testing procedures test main parts of the code, but do a poor job of testing extreme conditions. This

silent majority of the code comes into play when it is least expected and causes program crashes.

Software testing in its very simplified form can be reduced to the following three tasks:

- Generation of test-suites
- Validation of test-suite execution
- Regression testing

During the test-suite generation phase, programmers work on creating inputs to the program which force it to execute different parts of the code. The generated inputs are called test-suites. Good test-suites cover most of the code branches and fully exercise the program's functionality.

To validate the test-suite programmers run the program against the test-suite and check the output of the program to verify its correctness. If during this process errors are detected, they are fixed and the process proceeds until all tests execute correctly. At this stage the test-suite is ready for the third phase, which is regression testing.

In regression testing any new version of the program is run against the test-suite and its output is compared to the correct output. This phase makes sure that changes in the program do not introduce defects.

There have been continuous attempts to automate this process. Most success in testing automation has been in the automation of test case execution and result comparison, which is our phase three (Regression testing). There has been however, very little progress in automating the process of input data generation and test-suite validation. Any progress towards automation in these two areas is bound to bring huge payoffs:

- Test-suite automation will drastically reduce the cost of software development. Independent estimates [McConnell] suggest that cost savings could go up to 40%. This alone comes from the reduction of man-hours spent on test-suite generation.
- Automatic test-suite generation will significantly improve quality of the produced software. Complete testing of all parts of the software is not presently done, because test cases are hard to find or do not exist.
- Automation of test-suite generation will encourage programmers to test more code at early phases of the code development cycle, and in this way bring additional benefits through cost savings and improvements in quality.

One would expect that if the automation of test-suite generation can bring such a huge payoff there should be a lot of efforts and research in the industry to develop products which would fulfill this need. This is not the case. All previous attempts to generate such a

tool have been successful on small programs (up to few hundred lines of code) but all have failed on real life applications. Generally the problem is considered as very complicated and high risk. The complexity of the problem comes from its nature. To understand how difficult the problem is consider a typical 10,000 line program. When generating the test-suite, the programmer is trying to force the program to execute most of its statements. Statements in the program are executed as pieces of the execution path which the program is taking through its source code. In order to force the program through most of its statements, a significant number of paths need to be executed. In a 10,000 line program the number of paths can grow geometrically to 100,000,000. This number of paths is staggering, and is the reason the problem is so difficult to solve. Most people even consider it impossible to solve.

An attempt to build an Automatic Test Case generation is a high risk project. On the other hand, if the project is successful it is a breakthrough and the potential payback is huge.

The overall project goal is to develop a Test Generation System (*TGS*) which takes as input the source code of a program and ***automatically*** generates input for the program that satisfies a given coverage criteria (i.e. builds a test-suite for the program).

The main Phase I goals were to:

- Build a prototype of a *TGS* System which works with programs written in C. The prototype should have skeletons of the major algorithms which the tool will be using.
- Assess if the approach which we proposed has a chance to become a real *TGS* tool, by running the prototype on chosen C programs.

The goals of the Phase I were achieved. We produced a working prototype of the *TGS* tool. We implemented the major algorithms:

- Symbolic execution
- Dynamic Data-Flow Analysis
- Function Minimization Methods
- Heuristics
- Random Input Generation

We have tested the tool on three general purpose programs, `zip`, `flex`, and `tput`. All three applications are of medium size. `Zip` is about 9,000 lines of code, `flex` is about 8,000 lines, and `tput` is approximately 3,000 lines of code. The chosen applications are much larger than any applications for which automatic test case generation has been attempted. The biggest programs previously used as test cases were below 1,000 lines of C code.[Godzzila]. We consider it a significant achievement.

The results of our testing are encouraging. We were able to achieve, without human intervention, code coverage of 43% for `zip` 32% for `flex` and 32% for `tput`. These results are very good and suggest that the approach we chose has a good chance to lead to a tool which will be able to work on different types of real applications.

In the course of the research we have tested our algorithms. We discovered deficiencies in them and we believe that we have good idea how this deficiencies can be removed. We also found a lot of facts which we did not expect. All of the technical implications of our research are reported in section 5 "Lessons Learned".

In the course of our work we obtained answers to the questions which we posed in our proposal. Answers to them are distributed across of this report. Here we would like to give answer to the most important question which we asked "Is it possible to build an automatic input data generation system which can be successfully applied to real world applications?". We are confident that the answer to that question is YES. Yes it is possible to build the system which can work on large applications and we think that this report proofs it.

As we mentioned before were able to achieve these results only because we re-used a lot of code from our other programs. All code for parsing, flow analysis, and code reconstruction were re-used. We took them from commercial products that we've already developed. Only because of that were we able to build this prototype in such a sort time and concentrate on algorithm development instead of supporting software development. We do not believe that any other company is in such an opportune position to do the same.

2. Overview of the *TGS* System

The *TGS* system attempts to automatically generate test case data for a given program with the following criteria in mind:

- Full testing of all flow paths in the executable program.
- Identify inputs that cause the program to perform incorrectly.

The technique involves analyzing the program's code to identify points where input data is needed. The input instructions are then replaced with special functions which are able to simulate input using various techniques, ranging from random number generation to heuristic flow analysis techniques, to generate a set of input cases that satisfy the criterion above. The resulting program is then linked to a "test harness" which repeatedly runs the program, with varying input, adding unique test cases to a database of test case data. The algorithm converges when the criterion above are met, or no new test cases can be generated, in a reasonable amount of time.

Figure 1 shows the block diagram of an example program. Here it is seen that the program

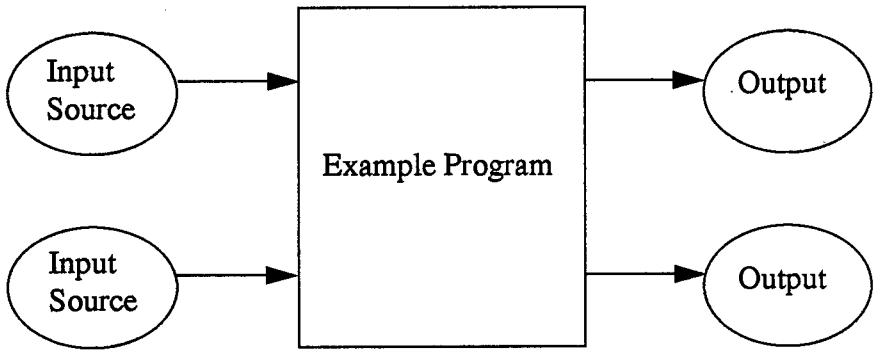


Figure 1: A Typical Program with Input and Output

takes its input from various input sources, such as a computer keyboard, or a file on disk, and generates certain outputs. In order to automatically generate inputs for this program the *TGS* System has to analyze the program's source code, find which instructions of the program correspond to the input, and replace them with code which can simulate input. This is done using the technique of source code instrumentation. We will describe the details of this technique in the following example:

```
void foo()
{
        char b, c, *ptr;
```

```
        c = getchar();
        if (c < '0' || c > '9') {
            *ptr = 0;
        }
        b = c - '0';
}
```

In this code, input is done through the function `getchar`. The programmer expects the input to be an integer ranging from 0-9. If the inputted value is not in that range, the program is designed to take the default operation, which in this case would not function properly. This is called an exception condition. Typically, however, exception conditions happen rarely, and the code that handles the exception is often times never tested, since in practice, exception conditions rarely occur. In this case the exception condition would cause the program to fail, since "`ptr`" is not yet initialized before it is de-referenced.

An internal representation of the program is shown in Figure 2. The boxes in the picture represent different nodes in the programs's "parse tree". Statements in the program are represented by one or several nodes in the parse tree. Each node has the field "next". This field points to the next node in the parse tree, or if the node is a leaf this field is 0. Consider the statement:

```
c = getchar();
```

This statement is represented by the "Assignment" node. The "next" element of that node points to an `if` statement. The "operands" element points to two operands of the assignment: "`c`" - left side and "`getchar`" - right side.

The parse tree is generated from the source code of the program through a process called parsing. The parser is a tool which is specifically designed to read the source code of a program written in a specific language and convert it into a parse tree. In our work we were able to use the standard ***ParaSoft*** C parser, and so we didn't have to develop a new parser.

The representation of a program in terms of a parse tree is unique and has a one to one correspondence to the original source code. In fact it is possible, and we used that technique, to generate source code out of the parse tree. The tool which does this is called a code reconstructor, and it works inversely to the parser.

It needs to be mentioned that the representation of a program in terms of a parse tree has several advantages. The most important one is that a parse tree is easy for computers to operate on. The second advantage is that a parse tree representation of the program is language independent, and any operations or tools which operate on it do not have language dependencies. This means that most of the work which we have done during this research is useful for other languages as well as C. In fact internal tools do not need to be changed when the prototype is modified to work with other languages.

Once the program is translated to a parse tree the *TGS* tool applies the code analyzer to locate points where the program is getting input. This allows the Tool Driver to choose inputs, at run-time, that will force the program to execute paths that may have previously never been tested. For the simple example in this case, if the input function "`getchar()`" returns any character whose ascii value does not fall within the numeric range (ascii 48 - 57), the program would fail. The test case generator, would then replace the input call to `getchar()` with a character generator, which could take random values. This would quickly lead to execution of all paths, after sufficient sample inputs have been tried.

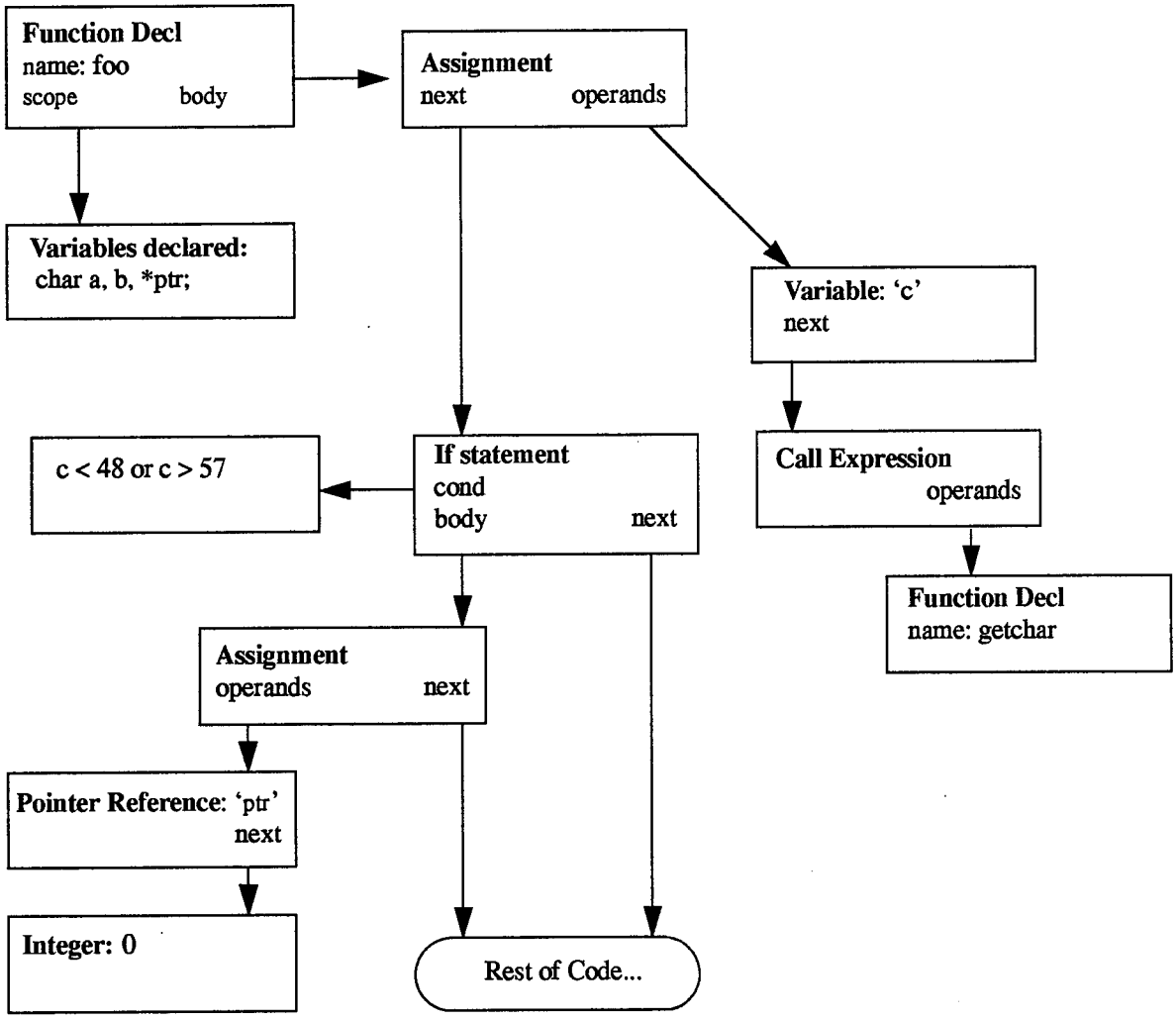


Figure 2: Intermediate Code for Untested Exception Handling Code

The code analyzer not only analyzes the code, but also instruments it to replace input functions with input generation functions.

Figure 3. shows the instrumented version of the code. As is denoted by the numbers, some

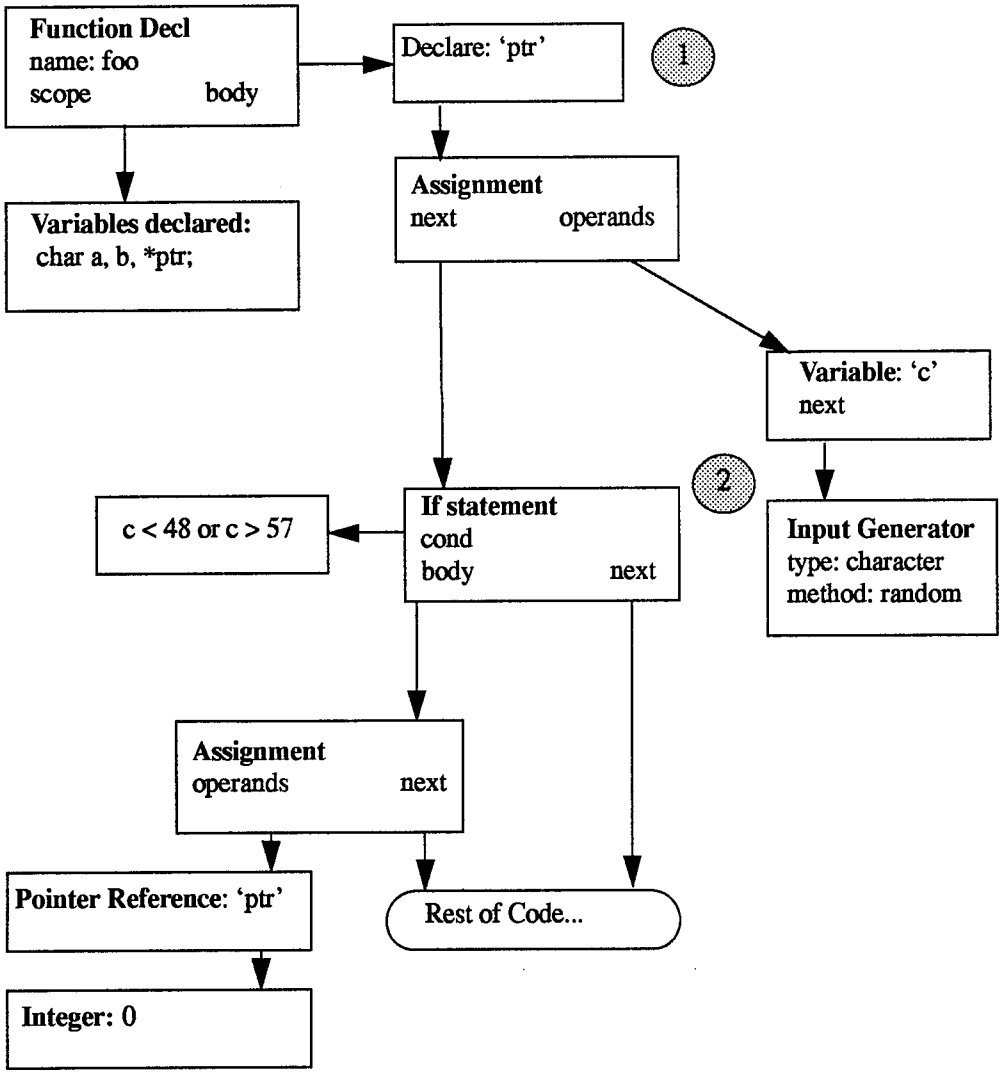


Figure 3: Instrumented Code with Test Case Generation Calls

code has been added to the code, and in one case, some code has been replaced. In order to have control over inputs that are received by the program, the call to `getchar()` is replaced by the Input Generator, which in this case generates random inputs between 0 and 255.

During the instrumentation phase, in addition to the instrumentation needed for input generation, two additional instrumentations are performed:

- Instrumentation for coverage analysis.
- Instrumentation for runtime error detection.
- Instrumentation for block flow.
- Instrumentation for branch condition analysis.

Instrumentation for coverage analysis is necessary. The information provided by it is used as a criterion for accepting or dismissing a test case.

The instrumentation for runtime errors is not needed. However it provides very helpful information. During of execution of the *TGS* system, the system can detect if the program hit an error and the user can be alerted to that situation. This is a very useful functionality. We did not need to do any extra work to achieve this, our instrumentation system did it anyway. Again in this case we received the benefit of re-using our existing code. On the other hand if the error detection functionality is not needed it can be turned of with a configuration switch.

The instrumentation for block flow monitors the flow of the program, stores it, checks if the desired flow is taken, and makes the decisions regarding continuing execution of the program or not.

In the instrumentation for branch condition analysis the branch conditions are replaced by an equivalent real valued function plus a call to the run-time that monitors its value.

The parse tree modified in this way represents an "instrumented parse tree". This tree is the passed to the code reconstructor to build the instrumented source code for the routine. The instrumented code is consequently passed to the language compiler The compiled version is then linked with the *TGS* input generation library into a final program ready for test generation.

During code instrumentation information about program flow, input statements, and branch condition form is stored in the database files. This information is used during the test generation process. It helps to decide which paths needs to be analyzed and how to generate input to force a program to take a specific path.

Once the program is generated the *TGS* tool driver will then run the program repetitively, using different inputs, until all possible paths in the program are executed.

During that process every set of inputs which is generated is analyzed by the "output analyzer". This tool checks to see if the input set increases the overall coverage of the program. If this is the case then the set is included in the test-suite. If it does not then the input set is discarded. In the current version of the tool when making decisions about the

input set we use block coverage criteria. This can be later changed and we can use other criteria, such as instance path coverage.

After the test case generator has run to completion, the test-suite now contains a set of test data. Each piece of data either results in the program taking a unique set of paths through the program, or results in a failed run of the program. Test data for which the program fails can then be analyzed, and the code corrected, so that future runs of the code are more robust.

Figure 4 shows the block diagram of the *TGS* System. the instrumented code for the above program., and the run-time support blocks that the instrumented code interacts with.

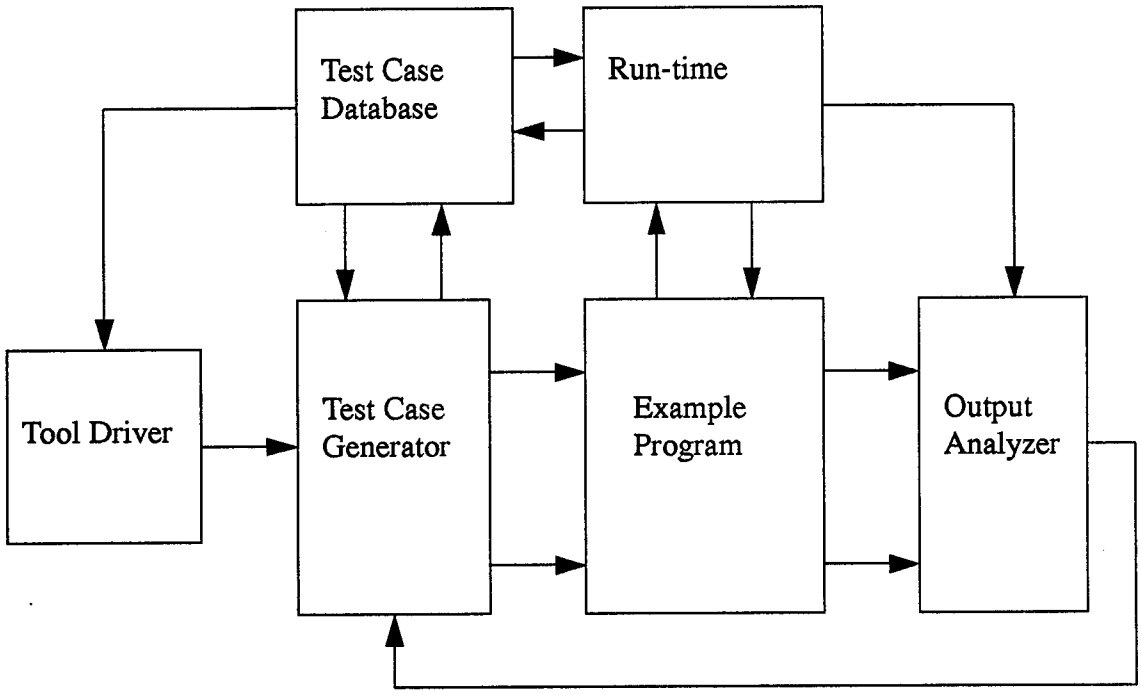


Figure 4: Test Case Generator Block Diagram

Additionally, the input sources have been replaced by the input generator which uses information from the output analyzer. As can be seen, the output analyzer uses output from the error checker as input, to decide if a run-time error was detected. The input generator uses different techniques to generate new test cases. The output analyzer evaluates the outputs of the original program, and that of the error-checker to evaluate the result from the current set of inputs. If an error is detected, or if at least one new flow path has been executed in the example program, this test case is added to the database. If the set of inputs

does not generate an error, or there are no new paths executed, it is discarded, and the system moves on to another test case.

3. Technical Description of the Prototype

A prototype for the Test Generation System (*TGS*) has been built. The prototype takes a C program as input and automatically generates input for the program that satisfies a given coverage criteria (i.e. builds a test-suite for the program). The prototype has been built with the following goals in mind:

- The system has to be as automatic and general as possible.
- The system has to be modular and expandable.

This is needed so that the system can be easily experimented with, with both different applications and input generation techniques. The prototype is built in such a way that it is a skeleton of the possible real system.

When developing the system we tried to minimize the dependency of the specific language, which is C. The language dependency is located in the code parser and code reconstructor. If these two modules are replaced, the system can work with any other language. The parse tree which we used is the standard ***ParaSoft*** tree format and it is used to support C, C++, Fortran77, and Fortran90. We also investigated the suitability of that tree to support Ada and we are confident that the tree structure is capable enough to support this language as well.

Modularity of the prototype is quite extensive. Modules interact with each other through well defined interfaces which are module independent. Replacement of modules and extensions to modules is easy and does not require modifications of other parts of the system.

First we will describe the basic functionality of the prototype and afterwards the different techniques we implemented to improve the generation of input.

3.1 Basic Input Generation System

The prototype generates test-suites using the two basic steps:

- Analysis and instrumentation of the original program.
- Execution of the instrumented program.

3.1.1 Analysis and Code Instrumentation

During the instrumentation phase the program is parsed and analyzed, the results of the analysis are stored in a database for the program, and an instrumented equivalent version

of the original program is generated. The instrumented C program is compiled and linked with a run-time library. The instrumented program has extra calls in it, the purpose of those calls is to:

- Automatically generate input for the program. The original input statements are replaced by calls to a run-time library that will generate the input.
- Record information about the program that is not available at compile-time. For example the flow taken by the program. This information is also stored in the database for the program.

This phase has been implemented by extending the functionality of ***Insight***, which is our run-time error detection tool. The ***TGS*** tool is implemented as a new switch "-Ztgs" to *insight*.

We will describe the operation of the analysis and instrumentation of the program on a simple example.Consider that we have a program which has its source code stored in files foo1.c foo2.c foo3.c. In order to build an executable from it the user normally performs following steps:

```
cc -c foo1.c
cc -c foo2.c
cc -c foo3.c
cc -c foo4.c
cc -o foo foo1.o foo2.o foo3.o foo4.o
```

Analysis and instrumentation is performed in a very similar fashion. The `cc` has to be replaced with `insight` and extra switch "-Ztgs" has to be added. Thus in order to prepare a program to work with the ***TGS*** system the user needs to change the previous commands to:

```
insight -Ztgs -c foo1.c
insight -Ztgs -c foo2.c
insight -Ztgs -c foo3.c
insight -Ztgs -c foo4.c
insight -Ztgs -o foo foo1.o foo2.o foo3.o foo4.o
```

Notice that the changes are minimal. The syntax of the changes was chosen to be of that form in order to minimize changes required in users makefiles. It should be noted that these type of changes can be done by changing the one line in the makefile which defines the compiler from `cc` to `insight`. For example:

`CC=cc` becomes `CC=insight`

In the above example the first 4 lines perform analysis and instrumentation on independent source files. During that phase files with `.c` extensions are converted to `.o` instrumented object files. Notice that at that stage the following actions were performed:

- Analysis of the program.
- Instrumentation of the parse trees
- Reconstruction of the instrumented source code.
- Compilation with the compiler.

At the same time the tool generates information database files with `.db` extensions. These files contain specific program information needed by *TGS* and flow analysis.

At the last step the programs object files are linked together along with the library which contains the input generation routines and run-time support routines for *TGS*. The resulting object code is ready to be used by the second part of the *TGS* system.

We would like to stress here that we designed the system in such a way that it will be very easy to use. The design is based on our past experience with over development tools. The system is designed to be used frequently. This requires a very simple user interface and easy modifications from standard compilation to compilation for the *TGS* system. This should encourage users to use it. It does not matter how useful a system is, if it's complicated and has an awkward user interface it is not going to be used.

3.1.2 Searching for the Test Cases.

In the second step the program is executed repeatedly. This step is completely automatic and done by a tool driver. The tool driver continues executing the program using the different input techniques and analyzing the results of the execution. It stops execution when it finds complete (100%) coverage or is told to stop by the user. The tool driver organizes the results obtained and produces the following output:

- A report showing the progress of the input generation. Among other things it has information about: the number of inputs generated, the program coverage they achieved, a summary of the actions taken, time spent, input generation techniques used, etc.
- A test-suite for the program consisting of the successful inputs generated, a set of inputs for which the program show bugs. i.e. inputs for which the program crashed.

We will continue description of the functionality of the prototype using the foo example which was begun in the previous subsection (3.1.1). The executable of the program ready to be executed by the tool is stored in the file `foo`. In order to start test-suite generation the user executes:

```
tgs foo
```

In this command `tgs` is the name of the tool driver. The tool takes several arguments and various flags. All possible options which can be passed to the tool are described in Appendix A which contains the `tgs` manual page.

Once the `tgs` command is executed the user is presented with the screen shown in Figure 5.

When *TGS* starts it reports information on where program was run and in which way it was invoked. Then it proceeds to generate test inputs. For every new input generated *TGS* reports:

- Number of runs - attempts to generate input. This number is split into three numbers: total number of runs, number of test cases saved in the test-suite, number of test cases which reported run-time errors and are saved in the "Bugs Test-Suite".
- Code coverage information.This is reported as three percentage figures: Code coverage of this specific test, total coverage of the program which is stored in the test-suite, total coverage including the test-suite and the Bugs test-suite.
- Action which was taken regarding the input. if the input expanded coverage of the program it is added to the test-suite. If the input forced program to encounter an error it is saved in the Bugs test-suite.

```
Test Generation System:
-------------------------------------------------
Directory: /home/lion2/quality/test
Command : tgs -heu foo
Date : Mon Nov 28 12:07:51 1994
Testbed : sun4c SunOS 2 4.1.3_U1
Host : lion
```

RUNS Tot in TS bugs	Coverage run TS Total	Action	Elapsed Time Run Total
1 1 0	3% 3% [3%]	added to test-suite	0:25 [0:30]
2 2 0	6% 7% [7%]	added to test-suite	0:18 [0:52]

Figure 5. Sample TGS Output

Otherwise it is discarded.

- Elapsed time of *TGS* execution in seconds. This is again reported as two numbers. The first reports the time spent executing the specific test case, the second number is the total elapsed time from the beginning of *TGS* execution.

During execution *TGS* generates several directories and files where the output from the tool is stored. In the current directory where the program was executed, *TGS* generates a `tgsdir` directory which contains the results of the runs. In this directory the tool creates a subdirectory "`ts`" which contains the generated test-suites. Inside the `ts` directory each test case is stored in a separate subdirectory named `t#` where # is the consecutive number of the test-case. Each `t#` directory contains a subdirectory `in` and `out`. The `in` directory contains input files which the program will use to run the test case. These files can be passed both to instrumented and non-instrumented versions of the program, to run the specific test-case. The `out` directory contains the output from running the specific case. The files are stored so that the user can check if the execution of the program was correct. They are not used in further parts of the system.

In addition to input files the *TGS* system generates an `rtest.scr` file in the `tgs` directory. This file is the script file which is used by the automatic test-replay tool "`rtest`" to automatically run the test-suites. "`rtest`" is our internal regression testing tool which we use to run our test-suites. Generation of the rtest.scr file is an extra benefit of using *TGS*. It automates the testing procedure to the highest possible extent.

3.1.3 Basic Modules of the Prototype

The prototype consists of four basic software units that are used by all of the input generation techniques. These units are:

- Compile-time unit: this is the unit that takes the original program, makes a static analysis of it, generates the instrumented program, and compiles it to produce the instrumented executable. The unit is based on *Insight* and it uses a lot of *Insight*'s technology. The other three parts are completely new modules and were developed specifically for this project.
- Run-time unit: this is the run-time for all instrumenting calls added by the compile-time unit:
- Tool-driver unit: automatic driver for the test-generation system, keeps executing the program while analyzing the input and resulting output and interacts with the run-time and the database for the program to apply

the different input generation techniques.

- Database unit: database of the program first created by the compile-time unit and accessed by all units.

The most effort during our research was put in the Run-time unit and Tool-driver part which interacts with input and chooses different algorithms for test-generation. These parts are also constructed modularly so we can add new test generation techniques and experiment with new algorithms. In this research we tested the feasibility of 5 techniques described in the following section.

3.2 Input Generation Techniques

This section describes the different input generation techniques we implemented. They correspond to increasing levels of sophistication and should allow us to see if automatic input generation is feasible. Each technique has only implemented the basic algorithm. Each of them is quite large and implementation of them in full detail is beyond the scope of this research. The main goal was to see if all or some of them in combination can be a basis for the creation of a real test generation system.

3.2.1 Random Input Generation

This is the most basic input generation used. It generates random input whenever the program requests any input. The values are random, but of the appropriate type. Input here and for all the other techniques is generated for anything except graphical input. For example the program may read from `stdin`, then open some files and read them, etc. The prototype in this case will create `stdin` and the other files and fill them with suitable values.

The prototype generates input while the program executes and at the same time creates the input files that would generate that input when running on the normal program. The run-time also generates an input description file which contains a detailed description of all the input generated for the program. The input description file is used by other input generation modules.

The run-time detects the data type of the input requested and generates a random value distributed uniformly over the range of valid inputs for that data type. It also decides randomly when to generate an end of file and when to generate inputs shorter than the requested ones (i.e. for `fgets` or `fread` system calls). The information needed by the run-time to generate the input is passed as parameter arguments to the run-time calls and through a database for the program that is created while processing the source.

3.2.2 Heuristics

The heuristics module controls which heuristics rules are activated while processing the program or running it. Any heuristics rule can be turned on or off independently. The current heuristics rules implemented work on the generation of input. Its purpose is to generate inputs that have a better chance to cover more parts of the program than the randomly distributed values generated by the random input generation module.

The rules for every kind of input (numeric or character/string based) have a relative weight. The system first decides randomly and according to the relative weight whether to generate input according to one of the active rules or randomly. Once the rule is decided, the system generates input according to that rule.

Examples of rules implemented for numeric values are:

- Exponential deviate: generates values distributed with an exponential deviate around 0.
- Uniform log values: generates values distributed uniformly in each order or magnitude. For example in that rule it has the same probability of generating 10, 100, 1000, ...
- Extreme values: generates one of a list of extreme values (0, 1, 2, -1, -2, ...).

Example of rules implemented for character/string values are:

- Extreme values: generates extreme values. For example if asking for a string generates the same character for each element of the string, or strings of zero length.
- Special values: generates one of a list of special values. Examples of special values currently included are C and basic keywords, minimal C, Fortran and Lex programs.

3.2.3 Function Minimization Methods

Here we have implemented the function minimization methods proposed by Korel [Kor90]. These techniques associate a real valued function to all the branches of the program. The problem of generating input so the program takes a given path is transformed with this technique to the problem of minimizing the associated *real valued function*. The *real valued function* allows us to use guiding techniques to find local minimums for the function. This technique is used by the tool driver to guide the generation of input.

Each input generated by the tool corresponds to a path in the program. Each successfully generated input is stored in a test-suite for the program. The run-time tracks the execution

flow of the program and checks at any point whether or not the current flow leads to uncovered parts of the program.

Given a path for the program for which we know the input, we generate input for a path that differs by taking an alternate branch in that path using function minimization methods. This allows us to generate more inputs for the program, given that we already have some input for it.

The function minimization technique we used works as follows:

The tool driver goes over an already existing path in the test-suite and traverses the path. At each selection statement in the path it checks if the branches other than the one taken by the existing path are already covered. If some are not covered, it tries to generate input so that the generated path is taken up to the selection statement and at that point the alternate branch is taken.

To do that it takes as a starting point the input for the already existing path. The idea is to modify this input so that the path up to the chosen selection statement is taken, but then the alternate branch is taken. This is accomplished by associating a *real valued function* to that branch. The *real valued function* is defined in such a way that the branch will be taken if the function becomes zero or negative. Thus the problem is transformed into a function minimization problem.

The tool driver then proceeds to execute the program repeatedly, monitoring the path taken by the program and the resulting value for the *real valued function* associated to the branch. While doing that it loops over all input variables and using *guided function minimization* methods to modify the input so that the function becomes zero or negative.

If the search succeeds the input that forces that branch is added to the test-suite of inputs for the program. If the search fails, the branch is marked as unfeasible. Afterwards the tool driver proceeds to find another uncovered branch in the existing paths in the test-suite. The algorithm can fail to find the input either because it takes too long or because the path condition gets broken (i.e. changing the input value makes the program take a path that doesn't lead to the selection statement we are concerned with.)

3.2.4 Dynamic Data Flow

Another technique proposed by Korel [Kor90] is to use dynamic data-flow along with function minimization. The purpose of this technique is to allow us to find out what specific input influence a given branch. The guided function minimization methods are thus optimized because the amount of input to try is reduced. Here we have implemented a variant of the method proposed by Korel. The tool driver monitors the actual flow for a given input of the program and calculates the data-dependence for that flow. This

information along with the actual values generated at every point in the program (also available to the tool driver) allows us to calculate dynamic data-flow information.

Dynamic data-flow techniques are used to reduce the space of input variables one needs to search when finding input for a given branch. The dynamic data-flow module is invoked when using the function minimization methods. After the tool driver decides to try to search input to force a given branch, it calls the dynamic data flow module to determine which inputs influence that branch. That information is given with reference to the input description file. The guided search method is then optimized because it only needs to change input values associated with the branch.

To calculate the dynamic data-flow information we used a variant of the algorithm proposed by Korel. In our tool we also wanted to include symbolic execution, so we used a generalized algorithm for symbolic execution that also allows the calculation of dynamic data-flow information. The symbolic execution algorithm allows us to calculate the data-dependence along any flow in the program. To calculate the dynamic data flow dependence it calculates the data-dependence along the flow actually taken by the program, and that leads to the branch condition we want to force. All this information is calculated every time the program is executed and is available to the tool driver because it is stored in the test-suite along with the input generated and the input description file. The tool driver combines the information of the data dependence for the actual path with the actual values found in the input description file to obtain the dynamic data dependence.

3.2.5 Symbolic Execution

Symbolic execution is a technique studied in the literature [Off91] to generate input for a given program. In this technique the program is executed symbolically and the condition for a given branch to be taken is transformed into some set of symbolic expressions satisfying specific conditions. This technique doesn't require the actual execution of the program. We chose to base our system on actual execution because we believe that actual execution is the only practical way of executing real programs. Symbolic execution has difficulties with standard constructs appearing in any appreciable real program (arrays, pointers, external functions, loops). Nevertheless whenever symbolic execution is possible it is a very efficient technique. For this reason we added symbolic execution to our prototype so we could use a mixture of both techniques and get the best of each of them. The compilation-time unit was extended to perform symbolic execution of simple constructs and to add the results to the program database. Using this technique, when trying to satisfy a given branch condition the tool driver first looks to see if symbolic execution was possible for that branch, if so, it uses symbolic execution techniques [Off91] to determine what input is needed to ensure that the branch is taken.

A symbolic execution is performed for the entire program while processing the source. To perform the symbolic execution the tool goes over all the source of the program and

calculates for all possible paths the path conditions and symbolic expressions for the variables. While going over non-selection statements the expressions in the program are used to obtain symbolic expressions for the relevant variables. Whenever a selection statement is found the path flows independently on every branch with a different path condition for each. When the paths flow again together at the end of the selection statement we have multiple paths flowing at the same time each with its own symbolic expressions and path conditions.

In real programs there are many constructs that prevent symbolic execution or make it inefficient. Some of these constructs are arrays, pointers, external function, loops. etc. In some of these cases the symbolic expressions and path conditions are not calculated and as a result we have unknowns for those expressions. The results of the symbolic execution are stored in the database for the program. We are particularly interested in the symbolic expressions for branch conditions along with the associated path conditions.

At the same time that the tool calculates the symbolic expressions it also calculates the data-dependence information. The data-dependence information is just like a symbolic expression but with missing information regarding the form of the dependence. The plain data-dependence information is simpler to calculate than the explicit symbolic expression and for some branch conditions the tool is able to get data-dependence information, but not the symbolic expression.

When using the function minimization techniques to force a given branch the tool driver looks in the database to see if a symbolic expression is available for the branch condition for that branch. If the symbolic expression information is available it uses it to directly deduce an input that will force the taking of that branch. If a symbolic expression is not available for that branch it uses the normal function minimization techniques described above.

3.3 Application of Input Generation Techniques.

During the course of our research we also tried to see if a combination of specific techniques leads to better coverage results. The particular technique which we used is described in the next subsection.

In the Phase I proposal we described only one algorithm which we wanted to apply to the test-generation problem. During the course of our research however, we realized that this is not possible, and that we need to implement each step of the algorithm as an independent module and then put them together as one of the possible options. The design which we have now is better than the one we originally proposed. The system is more expandable and ready to be tested with different algorithms.

3.3.1 The Combined Algorithm

The tool we have developed uses a combination of all the algorithms explained above to generate input for any given program.

The algorithms are used in the following way:

- The tool used random input generation plus heuristics rules to generate as many different inputs as possible for the program. The generated inputs are added to the test-suite along with all the information about the path taken and the input description file. Note that for any program the random input generation plus heuristics will always be able to generate at least one input.
- Once no more input is found by random input generation and heuristics, the tool switches to guided function minimization methods to derive new paths starting from the ones already in the test-suite. The tools loops over the existing paths and looks for alternate branches that are not covered. Once it finds one, it first looks to see if it has symbolic expression information for it, if so it used it to deduce the input needed so that the branch is taken. If no symbolic expression information is available for the branch condition it uses the function minimization methods along with dynamic data flow information to try to generate input to force that branch.
- Next the *TGS* driver starts the process of generating input variables. This process is carried out in a loop using information generated in the previous stages. The starting input set for the generation of input data is the one which was used for execution of the program. The tool goes to the end of the execution path and backups to the last branch. The information from the dynamic data flow analysis tells which input variables have influence on the branch condition in question. The tool then generates random inputs for only those input variables, and uses constraints from the symbolic execution of the program to see if the branch will be taken. If the satisfying set is found in the predefined time, the generated input set is added to the set of test inputs. If the solution is not found, the path is marked as not feasible.
- *TGS* next considers the path opposite the branch taken. If this branch does not have any more leaves, the tool backups to the branch above it. If the branch has leaves, the tool executes the program again with dynamic flow analysis to determine variable dependencies. If the branches below do not have loops or array accesses which influence branches, the run with dynamic flow analysis is not necessary, and is not

executed. The tool in this case attempts to generate new inputs randomly. The described process is repeated until inputs for all paths through the program are found.

The described algorithm is one of the possible combinations of the implemented modules. We only had the chance to test this one combination on real examples. The algorithm seems to work quite well.

During the course of our research we have learned that in some cases different variations of the algorithms may be needed. That is why we implemented the tool in a way that is easy to experiment with it and combine algorithms in different sequences.

4. Evaluation of the Created Prototype

We have run the *TGS* tool on different programs. Most of them were small test cases which we used to debug the code we developed. In addition, we also used the tool on three larger applications:

- `zip` - a popular program to compress files for more efficient storage on disks.
- `flex` - a common tool for generating programs which recognize lexical patterns in text.
- `tput` - simple program which uses curses library to put characters on the screen.

The choice of the programs was motivated by our desire to test different types of the applications.

`zip` is a typical numeric application which reads files. It is quite large, close to 10,000 lines of code. This code is much larger then we originally anticipated testing. In our proposal we expected that the prototype would be tested on codes between 1000-3000 lines. We performed some testing during development of the code and the results were encouraging enough for us to undertake much larger programs. The `zip` program takes only numeric input and this was in agreement with our work plan.

`Flex` is a common tool used by programmers. It reads an input file, parses it lexically, and based on that generates a file with C source code in it. We chose it because rather than using numeric input it uses lexical input, and we wanted to see how well our algorithms apply to generating lexical data.

In order to get some experience with graphical input and see how this type of input can affect our future research we also tested the `Tput` program. `Tput` is a much smaller program, only 3,000 lines of code. It is however in the upper limit of the programs we expected to test. The program uses the `curses` library to get input from terminals. `Curses` is a small graphics library and we chose it because it allowed us to get experience with graphical input without significant programming effort.

We also attempted to run the prototype on our own C parser which is a large program of about 200,000 lines of code. The result of the attempt was not successful. The prototype is not yet ready to handle such big programs and we stopped the attempt very early.

Before we describe the test results of running the prototype on the three bigger example programs we would like to describe the results of running the tool on a particular program - `long.c`. This program was specifically created by us to test different techniques as they were developed. It is a good starting point to show the benefits of different modes of the tool.

The `long.c` program demonstrates also another important thing. Its size is typical of programs which have previously been attempted with automatic test generation. In some sense the program demonstrates the current state of affairs when we started our research. Obviously tools which cannot work on bigger programs than `long.c` have no practical value in the real world. Our two other test cases `zip` and `tput` are real applications which are several orders of magnitude larger.

4.1 The long.c Program

The listing of the `long.c` program is included in Appendix B.

The program is very simple. It takes character input and either classifies it as bad input or prints it out. The program does not have a significant number of branches, but as can be seen from the test results, in order to achieve 100% coverage the *TGS* tool had to use all of its algorithmic arsenal.

We compiled the `long.c` program with `insight` and the `-Ztgs` switch and next we ran it under the `tgs` tool. As we described in a previous section the `TGS` system automatically uses different input generation techniques to build the test cases. First the tool tries the random and heuristic methods and afterward it uses the remaining techniques.

The output of the tool is shown in the following table. If we look carefully in the table we can see how the program worked. Up to step 5 the tool is using the random and heuristic algorithms. At that point it had generated 1 useful test case and 4 others which did not increase coverage. The tool then decides to switch algorithms. It starts using the guided algorithms and coverage begins to increase. The total coverage of the program increases to 94% which means that the entire program was covered.

In the Action column in table 1 new output starts to appear. First we see RVF = 40. This appears in guided mode when the tool is trying to find input to force a given branch. RVF = 40 means that the real valued function associated to that branch has value 40 for that run. *TGS* keeps rerunning the program with different input, looking at that value and based on

the changes in that value it tries to deduce input that will minimize that value and thus force the branch. "RVF = 40" is a shorthand for "Real Valued Function = 40".

RUNS Tot inTS bugs	Coverage run TS Total	Action	Elapsed Time Run Total
1 1 0	29% 29% [29%]	added to test-suite	0:01 [0:03]
2 1 0	29% 29% [29%]	discarded	0:01 [0:05]
3 1 0	29% 29% [29%]	discarded	0:01 [0:07]
4 1 0	29% 29% [29%]	discarded	0:02 [0:09]
5 1 0	29% 29% [29%]	discarded	0:02 [0:12]
6 1 0	11% 29% [29%]	RVF = 40	0:02 [0:15]
7 1 0	11% 29% [29%]	RVF = 41	0:01 [0:17]
8 2 0	35% 41% [41%]	added to test-suite	0:02 [0:19]
9 2 0	23% 41% [41%]	RVF = 107	0:01 [0:22]
10 2 0	23% 41% [41%]	RVF = 108	0:01 [0:24]
11 3 0	29% 47% [47%]	added to test-suite	0:01 [0:26]
12 3 0	17% 47% [47%]	RVF = 19	0:02 [0:29]
13 3 0	17% 47% [47%]	PC broken (exit)	0:01 [0:31]
14 3 0	17% 47% [47%]	PC broken (exit)	0:02 [0:33]
15 3 0	17% 47% [47%]	RVF = 18	0:01 [0:36]
16 4 0	47% 64% [64%]	added to test-suite	0:02 [0:38]
17 4 0	29% 64% [64%]	RVF = 58	0:01 [0:41]
18 4 0	29% 64% [64%]	PC broken (exit)	0:01 [0:43]
19 4 0	29% 64% [64%]	PC broken (exit)	0:02 [0:47]
20 4 0	23% 64% [64%]	PC broken (exit)	0:01 [0:49]
21 4 0	23% 64% [64%]	PC broken (exit)	0:02 [0:52]
22 4 0	29% 64% [64%]	RVF = 58	0:01 [0:55]
23 4 0	52% 76% [76%]	added to test-suite	0:06 [1:01]
24 5 0	35% 76% [76%]	RVF = 33	0:02 [1:04]
25 5 0	58% 88% [88%]	added to test-suite	0:01 [1:05]
26 5 0	41% 88% [88%]	RVF = 87	0:01 [1:06]
27 5 0	29% 88% [88%]	PC Broken (exit)	0:02 [1:08]
28 5 0	41% 94% [94%]	added to test-suite	0:02 [1:10]

Table 1: TGS output for long.c

The next interesting entry in the table is `PC broken (exit)`. This also appears only in guided mode. While *TGS* repeatedly runs the program changing the input, it can happen that for the new input the program just takes another path and doesn't get to the desired branch. When the run-time detects that the program deviates (before getting to the branch

in question) from the reference path it halts execution of the program. "PC broken (exit)" is a shorthand for "Path Condition broken (program exits)".

The table shows that the runs did not take very long. The total finding process took about 1 minute. This is not typical and our experience shows that for larger programs the searching process may take much longer. This can be seen in the next two test cases.

4.2 Test Results for Zip

The next program which we ran the tool on was `zip`. Zip is a popular program used to compress files. It reads as input a file and as output produces compressed file of smaller size. As we mentioned before this program is about 10,000 lines of code. Being able to run the prototype on a program of that size is a real success. To the best of our knowledge nobody has been able to automatically generate test cases for programs of that size. This was considered impossible! Being able to do that is a potential break-through. It proves that the algorithms we designed have a significant potential to work on real applications. However we have to be very cautious here.

The program was tested in the same way as before. We compiled the `zip` program using `insight` with the `-Ztgs` flag. The program consisted of several files which were first compiled, instrumented and linked together. After the program was linked we ran it with the *TGS* tool.

During compilation and early runs of the program we encountered many problems. We discovered a lot of bugs and inefficiencies which prevented us from running the tool. Our first results of running the tool gave us only 5% coverage. This was very low and we worked hard to improve it. At the end we were able to achieve coverage of about 43% which we think is quite significant for that type of program.

The coverage of 43% on `zip` does not mean that if the tool is used on a similar program of that size the same results can be achieved, probably not. The tool is a prototype and requires a lot of work to reach that stage.

The results of running *TGS* with `zip` are reported in table 2. The table stops after 16 tries.

RUNS Tot inTS bugs	Coverage run TS Total	Action	Elapsed Time Run Total
1 1 0	3% 3% [3%]	added to test-suite	0:17 [0:36]
2 2 0	6% 7% [7%]	added to test-suite	0:09 [0:50]
3 3 0	25% 29% [29%]	added to test-suite	0:17 [1:11]
4 4 0	32% 36% [36%]	added to test-suite	1:03 [2:18]
5 5 0	6% 36% [36%]	discarded	0:08 [2:31]
6 4 1	1% 36% [36%]	added to bugs	0:11 [2:47]
7 4 1	14% 36% [37%]	discarded	0:10 [3:02]
8 4 1	31% 36% [37%]	discarded	2:51 [6:00]
9 4 1	31% 36% [37%]	discarded	2:30 [8:30]
10 5 1	32% 37% [38%]	added to test-suite	2:40 [9:04]
11 6 1	8% 38% [39%]	added to test-suite	0:20 [9:28]
12 7 1	2% 39% [40%]	added to test-suite	0:02 [9:31]
13 7 1	6% 39% [40%]	discarded	0:08 [9:40]
14 8 1	33% 41% [42%]	added to test-suite	3:15 [12:48]
15 8 2	6% 41% [42%]	discarded	0:09 [13:00]
16 8 2	1% 41% [43%]	added to bugs	0:02 [13:02]

Table 2: TGS output for zip

After that we continued to run the tool for many hours but did not see any improvement. We analyzed why the improvement does not happen and our findings are summarized in the *Lessons Learned* section.

The table illustrates several things which we saw in many cases.

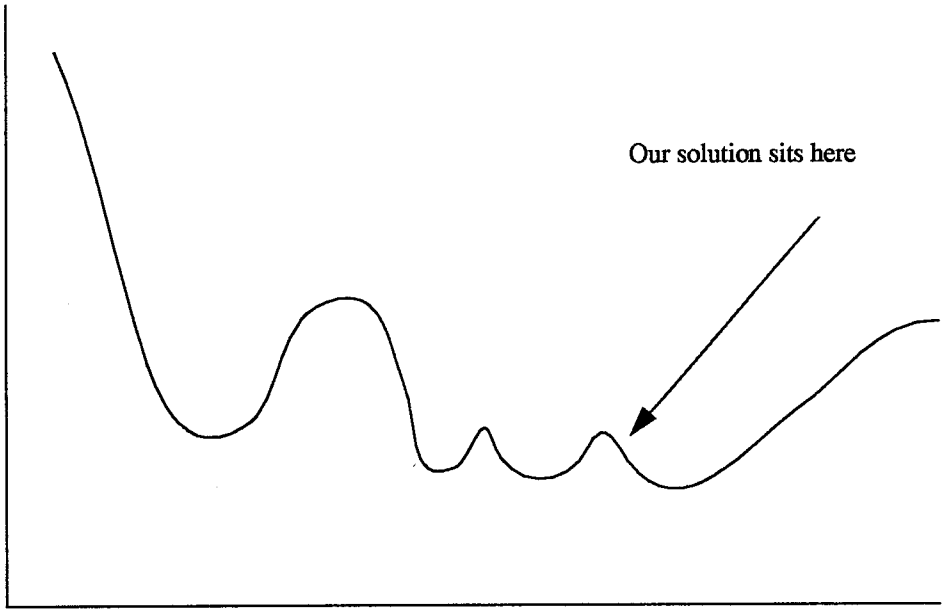
- At the beginning of test generation the tool finds new test cases relatively quickly. As execution progresses it becomes more complicated
- Switching algorithms leads to new inputs being discovered
- Some of the new runs differ only by one branch and even though the total coverage is increased it does not improve the global coverage of the program

The `zip` program confirmed one of our major thesis. We started this project with the hope that it will be possible to achieve incremental improvements by forcing the tool to switch branches and in this manner to cover new paths through the program. We analyzed this in the case of the `zip` program and we see that this is really what is happening. For example the tool obtained input for run 9 based on the input from run 8. We can see that the coverage

of these cases is very similar. In fact we checked it using a special tool developed for that purpose. We built a path visualization tool db, which can display paths taken through the program. The paths of case 8 and 9 differed by only one branch, the branch which was switched. Unfortunately that branch was already covered by the other paths and coverage for the run did not increase.

As we mentioned before, after 16 cases *TGS* was unable to detect any new input cases which would increase coverage of the program. We think this happened because none of our algorithms could start generating solutions in new areas. This was a big disappointment for us. It means that we have a lot of real work ahead of us in order to improve that behavior.

The result of not finding input can be explained with the following example. Our method of finding input can be described as a mathematical method of finding minima in a complicated function. Lets assume that the function looks like the following figure. The



whole *TGS* system can find solutions which do not differ a lot from each other, (the arrow points to that area). The tool however does not have a mechanism to jump over the bump on the left side and find a different solution. This is one of the things for which we do not have an algorithm and we do not know if the algorithm can be found at all. Nowhere in the literature we reviewed was that problem discussed. In future research we will have to find a solution to that problem if we want the tool to be successful.

4.3 Test Results for Flex

Another program we tested was the `flex` program. We chose the `flex` program as a test because it is typical of programs commonly in use in the industry. The `flex` program is a variation of the industry standard `lex` program. `Flex` is a lexical analyzer. By choosing `flex` we were trying to see how our techniques can work on programs which require cohesive input.

Again we compiled the program and ran it with the *TGS* tool (see table 3). The total coverage achieved here is about 32%. This coverage shows a significant benefit to the user. Actually we were surprised that the coverage is so high. We expected that for such a programs it is important that the input is generated according to the language rules. We thought that generation of the input based only on the source code of the program would not be enough. The results of testing `flex` were a pleasantly surprising.

The results of the *TGS* run for `flex` are shown in the table below:

RUNS Tot inTS bugs	Coverage run TS Total	Action	Elapsed Time Run Total
1 1 0	11% 11% [11%]	added to test-suite	0:32 [0:32]
2 2 0	10% 14% [14%]	added to test-suite	0:17 [0:49]
3 3 0	28% 32% [32%]	added to test-suite	0:53 [1:42]
4 3 0	11% 32% [32%]	discarded	0:41 [2:23]

Table 3: TGS output for flex

We can see that similar to the `zip` program the test cases are found most successfully at the beginning of a *TGS* run. Continuing the run is wasted effort.

In order to improve coverage for this program our analysis indicates that it will be necessarily to build input generators which have some knowledge of the syntax with which input should be generated. We discuss this issue in more detail in section 5.

4.4 Test Results for Tput

As a final test we tested the `Tput` program. As we mentioned we chose the `tput` program as our test, because it had some simple graphics routines. We wanted to see if it is possible to build a tool which can generate input for programs which use graphical input. We also wanted to see how we can work with external functions, and the graphical input functions were a good candidate.

Again we compiled the program and ran it with the *TGS* tool (see table 4). The total coverage which we achieved here is about 32%. This coverage is not as high as we would like, but it is encouraging.

The results of the run of *TGS* are shown in the table below:

RUNS Tot inTS bugs	Coverage run TS Total	Action	Elapsed Time Run Total
1 1 0	13% 13% [13%]	added to test-suite	0:40 [0:56]
2 2 0	8% 21% [21%]	added to test-suite	0:09 [1:05]
3 3 0	25% 30% [32%]	added to test-suite	0:17 [1:23]

Table 4: TGS output for tput

Again in this case the test cases are found at the beginning of the run and the rest of the run is wasted. This test shows similar results to the previous ones. Again we are caught in the global minimum and cannot get out of it. The results also show a different effect. External functions complicate the picture and it is easier for the tool to be caught in the minimum.

We believe that this result gives us two messages:

- It is possible to generate input to programs which have external functions.
- Our algorithms are not yet good enough to get significant coverage when program uses external functions.

We will have to put significant research effort into solving that problem. As we mentioned before most real applications use graphical input and it is imperative for the tool to work with such applications.

5. Lessons Learned from Phase I Effort

While executing this research we worked according to our work plan. Now looking back we think that we did not have enough time to analyze our prototype in full. We have done only preliminary investigations, but even they have taught us some very valuable lessons.

We think that the random + heuristics part of the tool seems to scale well. This means that it works well on large programs. This method however does not yield high coverage.

To get more coverage we proposed a more advanced algorithm which is a combination and extension of the known algorithms. This algorithm seems to be very promising but it requires a lot of work to get good coverage and to be able to scale to large applications.

However in our research we were able to achieve with our algorithm things which other people considered impossible. In the field it has been considered impossible to find input for programs larger than 1000 lines of code. We achieved that goal easily.

What follows now is a list of algorithmic problems/deficiencies we found and which we need to solve.

5.1 Guided Function Minimization Method

We implemented the guided function minimization methods as proposed by Korel [1]. They proposed for branch predicates being simple relational expressions. They need to be extended to cover any possible construct found in a C program. In particular they need to be extended to:

- Any branch predicate expression: one needs to be able to convert any branch predicate into a real valued function. They have been proposed only for branch predicates being simple relational expressions. We will need to extend it to any branch predicate expression. In particular it needs to be extended to branch predicate expressions being combinations of logical `AND` and `OR` expressions. For this case the only solution would need to be to define instead of just one function, a set of real valued functions and minimize them individually until the `AND` and `OR` expressions between the real valued functions are satisfied.
- Apply function minimization methods to `SWITCH` constructs.
- Extend the function minimization methods to loop constructs. Here we want to extend the guided function minimization technique so it can be

used to force the execution of loops a given number of times.

- Find faster algorithms for searching for the function minimum. The algorithm proposed in the literature was the alternating variable method [12]. We found experimentally that this algorithm requires too many iterations to find the minimum of the function in most cases. As each iteration required the execution of the program it is very important to find algorithms that progress towards the minimum in very few iterations. To solve this problem we propose to make assumptions about the form of the branch construct and use those assumptions to design search methods that progress towards the minimum very quickly. This require us to make a study of the branch constructs that appear in real programs and identify the constructs that appear most often.

5.2 Dynamic Data Flow

The main problem found with dynamic data flow is that it is expensive to calculate the dynamic data flow information as it takes a real execution of the program every time. Here we propose as a solution to do an exhaustive static data dependence of the program and use that information to deduce invariants that may allow us in most cases to calculate the dynamic data dependence without re-executing the program.

Another problem found with dynamic data flow is that even though it appreciably restricts the input space one has to search, it is still sometimes difficult to find input. The problem is that in most cases the path condition gets broken. To solve this problem we propose two solutions:

- Relax the condition in the path-condition, i.e. allow the program to execute even if the path condition is broken as long as it is in a path that can lead to the block we are trying to cover.
- Reorganize the searching on the set of input that influences the branch. For example search first over the input that influences fewer branches in the path.

5.3 Symbolic Execution

The symbolic execution techniques did show quite a few problems when applied to large programs. Whenever symbolic execution is possible it is very helpful. The problem is that there are many constructs that inhibit symbolic execution (array elements, pointers, loops, external functions ...). As the program size grows the probability of finding those constructs in any path grows as well, resulting in an inability to find correct symbolic expression forms for most branch conditions in a big program. To solve this problem we propose a novel technique that we call dynamic symbolic execution. This technique will consist in doing a standard symbolic execution for the actual path taken by the program. This will remove most problems involved with symbolic execution, like array indices used or number of times a given loop has been executed. This technique will be computationally intensive but should be very powerful. It will be used to solve the final hard branches in a program that cannot be solved with other techniques. With this technique one will get a symbolic expression, with very specific information about its dependence on input, for example if an array element appears on the symbolic expression it will give information about what its actual indices are and, if for example the array element is read in a loop, in which loop iteration that array element was read.

6. Potential Future Developments

In this section we would like mention only some of the things which need to be done to the prototype to make it really useful. We would like to stress that the prototype is a very simple program and its abilities are restricted. However it is a good foundation to continue future development. It also enables us to study the chosen algorithms and learn about their deficiencies. We now we have a much better picture of what needs to be done to build a tool which will work on real applications.

Our research teaches us that there are still a lot of unknowns, and that the road to a real tool may still be very risky. The algorithms which we used worked on the test cases, but they failed on our real product. There is a lot of algorithmic research which we will have to conduct to get the tool working on programs of 200,000 lines.

Actually we have learned that it is much more difficult to make progress from 10,000 line programs to 200,000 line programs than it is from 100 line programs to 1,000 line programs.

In the previous section we described the algorithmic modifications and research which absolutely needs to be done to make the tool more usable. Here we list some other extensions which we think would be very useful.

- Extend the tool to support other languages (C++, Fortran, Ada, ...). Most of the algorithms used are language independent. Nevertheless a few modules are language specific and will need to be written.
- The tool should be extended so it addresses the generation of input that covers only specific portions of the program. For example, parts that the user has modified or modules that have been added to an existing program.
- Accessing of the parse-tree structure by the run-time. It would be very helpful if the executing program was able to access its own parse-tree structure. In the future we would like to modify our parser and run-time to allow this possibility. Currently all the needed parse-tree information is mostly passed through arguments in calls to the run-time. But this method is not feasible in general, for example to pass the information about the elements that comprise an arbitrarily complex abstract data type, and is currently preventing the application of some of the input generation techniques to some constructs.
- Try different branch and path selection criteria. Given a path in the test-suite the tool attempts to generate input for derived paths obtained by forcing un-taken branches in the original path. The paths generated

this way cover all of the program in most cases. Nevertheless sometimes some portions of the program cannot be covered in this way. For this cases we will need to develop a technique that generates specific paths that cover those parts of the program remaining.

- Add a `lex`/`yacc` grammar generator to the tool. An appreciable number of programs use `lex` and/or `yacc` to process their input. It is very difficult to create input that conforms to specific `lex` and `yacc` rules. Adding a module that generates input according to `lex` and `yacc` rules will be a very big improvement for those kinds of programs.
- Convert the prototype into a production level tool. The prototype developed combines many algorithms and uses complicated techniques that were first experimented with on real programs using the prototype developed. There is a lot of work needed to apply the prototype to many different applications and optimizing/tuning/modifying the algorithms for the different issues that appear in real programs.
- The developed tool uses branch coverage criteria. The tool should be extended so it can address other testing criteria.

We think it would be very important to extend the tool so it is able to do module testing. In module testing one tests every module of the program (i.e. every procedure individually). To do this correctly and in a way convenient to the user will require a lot of work. Nevertheless we think it could be a very powerful technique complementary to that of testing the program as a whole. The main reason is that the work required to test a program using module testing will increase linearly with the size of the program. This will allow a very exhaustive test of all modules even for very big programs. Some of the ways in which the tool will need to be extended to do module testing are:

- Make the tool able to identify the units into which the program can be decomposed and extract them (i.e. each procedure plus all the related externals it needs).
- Write a module that is able to exercise those modules repeatedly until a given coverage criteria is achieved.
- Extend the input generation routines so they can produce the input required by the modules. For example a routine may accept a pointer to a structure which the routine treats as being the beginning of a linked list. The tool will be able to detect that and generate as input to the routine arbitrary linked lists.
- Whenever the tool detects a module with a problem, present to the user the source of the problem in a manner in which he can identify where

the problem comes from. I.E. It should be able to tell the user the form of the input that produced the problem, and allow the user to step through that routine with the debugger and with the input that originated the problem.

- Create a database with all the routines and the input that produce the required coverage criteria. This database will be used to re-execute the tests whenever the user wants.

We have mentioned here some possible future developments. The work which needs to be done is much more extensive and complicated. There is a lot of work which needs to be done and there are a lot of unknowns. The research is also very risky and many people have already attempted to pursue it and failed. We believe that with our ideas we have a chance to conquer these problems.

7. References

[1] [Bic79] J. Bicevskis, J. Borzovs, U. Straujumus, A. Zarins, E. Miller, "SMOTL - A system to construct samples for data processing program debugging, "IEEE Trans. on Software Eng., vol. SE-5, No. 1, Jan. 1979, pp. 60-66.

[2] [Bir83] D. Bird, C. Munoz, "Automatic generation of random self-checking test cases," IBM Systems Journal, vol. 22, No. 3, 1983, pp. 229-245.

[3] [Boy75] R. Boyer, B. Elspas, K. Levitt, "SELECT - A formal system for testing and debugging programs by symbolic execution," SIGPLAN Notices, vol.10, No.6, June 1975, pp234-245.

[4] [Cla76] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Trans. on Software Eng., vol. SE-2, No. 3, Sept. 1976, pp. 215-222.

[5] [Cla78] L. Clarke, "Automatic Test Data Selection Techniques," Infotech State Of the Art Report on Software Testing, September 1978.

[6] [DeM87] R. DeMillo, W. McCracken, R. Martin, J. Pasafiume, Software Testing and Evaluation, The Benjamin/Cummings Publishing Company, 1987.

[7] [Gil 74] P. Gill, W. Murray, Ed., Numerical Methods for Constrained Optimization, New York: Academic, 1974.

[8] [Gla65] H.Glass, J. Cooper, "Sequential search: a method for solving constrained optimization problems," Journal of ACM, vol. 12, No. 1, Jan. 1965, pp. 71-82.

[9] [Har88] M. Harrold, M. Soffa, "An incremental approach to unit testing during maintenance," Proceedings of the Conference on Software Maintenance, Phoenix, AZ, October 1988, pp. 362-267.How77] W. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Trans. on Software Eng., vol. SE-4, No. 4, 1977, pp. 266-278.

[10] [Inc87] D. Ince, "The automatic generation of test data," The Computer Journal, vol. 30, No. 1, 1987, pp.63-69.

[11] [Kor85] B. Korel, J. Laski, "A tool for data flow oriented program testing," SoftFair II, Proc. of 2-d Conf. on Software Development, Tools, Techniques, and Alternatives, San Francisco, Dec. 4-5, 1985, pp. 34-38.

[12] [Kor88a] B. Korel, J. Laski, "Dynamic program slicing," Information Processing Letters, vol. 29, No. 3, October 1988, pp. 155-163.

[13] [Kor88b] B. Korel, "PELAS - Program Error Locating Assistant System," IEEE Trans. on Software Eng., vol. SE-14, No. 9, Sept. 1988, pp. 1253-1260.

[14] [Kor89] B. Korel, "TESTGEN - A Structural Test Data Generation System," Sixth International Conference on Testing Computer Software, Washington D.C., May 22-25, 1989.

[15] [Kor90] B. Korel, "Automated Software Test Data generation," IEEE Trans. on Software Eng., vol. SE-16, No. 8, August 1990, pp. 870-879.

[16] [LaKo83] J. Laski, B. Korel, "A data flow oriented program testing strategy," IEEE Trans. on Software Eng., vol. SE-9, No. 3, May 1983, pp. 347-354.

[17] [Leu89] H. Leung, L. White, "Insights into regression testing," Proceedings of the Conference on Software Maintenance, October 1989, pp. 60-69.

[18] [Muc81] S. Muchnick, N. Jones, Ed., Program flow analysis: Theory and Applications, Prentice-Hall International, 1981.

[19] [Ram76] C. Ramamoorthy, S. Ho, W. Chen, "On the automated generation of program test data," IEEE Trans. on Software Eng., vol. SE-2, No. 4, Dec. 1976, pp. 293-300.

[20] [Wey85] E. Weyuker, S. Rapps, "Selecting software test data using data flow information," IEEE Trans. on Software Eng., vol. SE-11, No. 4, April 1985. pp. 367-375.

[21] [Woo79] J. Woods, "Path Selection for Symbolic Execution Systems," Ph. D. Thesis, University of Massachusetts, August 1979.

[22] [Off91] A. Jefferson Offutt, "An Integrated Automatic Test Data Generation System", Journal of Systems Integration, 1, pp. 391-409, 1991.

[23] [McConnell] Code Complete - Microsoft Press

[24] [Man94] Steve Mansour, "Vista vs. Hindsight", Advanced Systems, November 1994 pp. 43-47

Appendix A: TGS Manual Page

TGS(1)

NAME

`tgs` - Test Generation System

SYNOPSIS

```
tgs [options] command
```

DESCRIPTION

`tgs` is the driver for the Test Generation System (*TGS*). The *TGS* system is able to automatically generate input for the execution of any C program.

'`tgs`' keeps executing a program repeatedly and generating different input for every execution. The results of the execution of the program with the generated input are analyzed.

If some error is found during execution (i.e. program core-dumps or the insight runtime finds some problem), the input that caused the problem is stored in a subdirectory in "`tgsdir/bugs`". There is a subdirectory for every input that caused a different problem.

If no errors are found and the generated is added to the test-suite in "`tgsdir/ts`". There is also a subdirectory here for every input.

OPTIONS

`-check` Only checks for self-consistency of the generated input. Generates one input for the program, re-executes the program with that input, and checks that the output files are identical.

`-ddf` Use dynamic data flow methods.

`-guided [num]`
Enter guided function minimization methods after 'num' consecutive runs are discarded in random plus heuristics mode.

`-heu` Use heuristics to generate input for the program.

`-n runs` Generate at most 'runs' inputs for the program. If this flag is not set '`tgs`' keeps running until it achieves 100% coverage with the combined inputs in "`tgsdir/bugs`" and "`tgsdir/ts`".

`-pw` Use path-wise methods. Stops execution of the program as soon as it enters a flow that can lead only to covered parts of the program.

`-rtest` Do not write to '`stdout`' any output that can change between identical invocations of the same '`tgs`' command (i.e. timing information).

`-se` Use symbolic execution methods.

`-show_flow`
While the program executes, dumps a file ("`tgsdir/ts/t#/flow`") that shows the flow taken by the program.

`-stdin` Writes to the output the input that was generated for the program.

`-stdout` Writes to the output the output produced by the program.

`-stderr` Writes to the output the error produced by the program.

`-T` Timeout for the whole execution.

`-t` Timeout for each trial run.

`-_stop #`
Internal, for debugging. Stops 'tgs' just before executing run number #. Then the run can be executed under a debugger.

`command` The command that needs to be used to execute the program. Special shell characters ('>', '<', ...) need to be protected.

USAGE

To use the *TGS* system the program should first be compiled with '`insight`' using the "`-Ztgs`" flag. The resulting executable is then executed under the *TGS* system using the '`tgs`' command. All of the information generated by '`tgs`' is put under a subdirectory called "`tgsdir`" of the directory where the program was run. (See the section **FILES** for a descriptions of the contents of that subdirectory.)

FILES

`$TGS/.insight:`
The commands in this file should be in the "`.insight`" file in the directory where '`tgs`' is invoked.

`$TGS/demo:` This directory contains a very simple demo that shows how to use the system. To see it just type '`make`' in that directory.

`tgsdir:` This file is created in the directory where '`tgs`' is invoked. All files and reports created by '`tgs`' are put under this directory.

`tgsdir/REPORT:`
This is the report of the things done by '`tgs`'. It has information regarding the number of inputs generated, and the number of inputs included in the "`ts/`" (test-suite) and "`bugs/`" directories. This file is an exact copy of the information that goes to '`stdout`' while running '`tgs`'.

`tgsdir/ts:` Directory containing the test-suite generated by '`tgs`'.

`tgsdir/ts/t#:`
Here '#' goes from 1 to the number of inputs included in the test-suite. For each generated input included in the test-suite a directory is created to store it and other relevant information.

`tgsdir/ts/bflow:`
Recording of the actual flow taken by the program. Can be visualized using "`db -show_bflow bflow`".

`tgsdir/ts/t#/in:`
Directory containing the files that constitute the input for test case #. If input was needed from '`stdin`', the input generated is stored in a file called "`_stdin_`".

`tgsdir/ts/t#/out:`
Directory containing the files that constituted the output for test case #. '`stdout`' and '`stderr`' are stored in files "`_stdout_`" and "`_stderr_`".

`tgsdir/ts/t#/record.idf`
Input description file for the run. Can be visualized using "`db -show_idf record.idf`".

`tgsdir/ts/t#/report:`
Short report with some information about the I/O operations done by the program while executing (i.e. files opened and where).

`tgsdir/ts/t#/tca.log:`
Coverage logfile corresponding to the run in that directory.

`tgsdir/ts/t#/tca.out:`
Coverage summary for the run in that directory.

`tgsdir/ts/t#/inst.tgs:`
Instructions from '`tgs`' to the 'tgs-runtime'.

`tgsdir/bugs:`
Directory containing the input generated by '`tgs`' that caused bugs in the program. I.E. the program core-dumped or `insight` found a problem with it. The directory structure is the same as for "`tgsdir/ts`".

`tgsdir/rtest.src:`

Script for '`rtest`' to run the tests in the test-suite in "`tgsdir/ts`". It should be executed as "`rtest -s tgsdir/rtest.src`".

`tgsdir/tca.out:`

Coverage summary for all the inputs generated.

`tgsdir/dead_path:`

When running `tgs` with `-pw` it writes the file and line where a run was stopped because it entered a flow that leads only to covered parts of the program ("disc (dead-path)").

`tgsdir/ts/t#/flow:`

Flow that the program took while executing. This file is generated if '`tgs`' is invoked with the flag '`show_flow`'. Note that this file can be very big, so use this with caution.

For each `tca` block entered the following info is printed:

{tca_block_#}-file:line [c1c2c3] [c4].

c1, c2 and c3 are either a character ' ' or '*', a '*' means in positions:

- c1: this block was already covered in a previous run.
- c2: all the externals referenced in the block were fully covered in previous runs.
- c3: all the blocks reachable from this block in the current function were already covered in previous runs.
- c4 is '^' if the current run increases the coverage, otherwise is ' '.

`*.db:` for each source file in the application there is a corresponding `.db` file which has needed information from the file. The file is stored in the same directory where the source file is. If that is not possible the file is stored in the directory where '`insight`' was invoked from.

`db.ind:` list of files that constitute the application. There is one entry for each file included in the application (including included files).

`tca.db:` file with `tca` information for all of the runs generated by '`tgs`'. This file is written by a modified version of '`tca`' invoked with the flags '`-c -_db`'. The information in this file is used to update the information in the `.db` files.

`$TGSDIR/report.tgs:`
For each invokation of '`tgs`' a line is added to this file with a summary of the application name, coverage obtained, runs tried, time spent executing, and optimizations used. This file is only written if `TGSDIR` is set.

Appendix B: Code of long.c program.

```
/* Program that shows the LONG-INPUT problem. Coverage is only done if the
input string is "B ... E\n". */

#include <stdio.h>
 #include <strings.h>

static char input_string [1000], *c = input_string;

static void bad_input ()
{
    if (*(c-1) == EOF) {
       printf ("Premature EOF\n");
    } else {
      *c = '\0';
      printf ("bad_input = [%s]\n", input_string);
    }
 exit (1);
}

main ()
 {
   if ((*c++ = getchar ()) != 'B') bad_input ();
   if ((*c++ = getchar ()) != ' ') bad_input ();

   while (isalpha (*c++ = getchar ())) ;

   *(c-1) = '\0';
   printf ("input_string = [%s]\n", input_string);
   *(c-1) = ' ';

   if ((*c++ = getchar ()) != 'E') bad_input ();
   if ((*c++ = getchar ()) != '\n') bad_input ();
   if ((*c++ = getchar ()) != EOF) bad_input ();

   *(c-2) = '\0';
   printf ("Correct program = [%s]\n", input_string);

   exit (0);
}
```