Chapter 4

TSR Software Tool

4.1 TSR Overview

Based on the algorithms for the reduction of requirement-based test suites using EFSM dependency analysis described in Chapter 3, the *Test Suite Reduction* (TSR) program has been developed as a part of the Test Suite Generation/Reduction Software tool (TSGR) [20], which is implemented in C++ and Java languages and runs on Sun workstations under Solaris Sparc 5.8. Besides the Test Suite Reduction program TSR, there exists another part of TSGR called the *Reduced Test Suite Generation* (TSG) program, whose function is to generate a small yet effective set of executable test cases such that it exercises all possible SIPs w.r.t. each requirement under test at least once.

TSR is built to reduce the number of test cases in a given test suite by eliminating equivalent test cases, according to the definition of either SIP or DIP. TSR is made up of two sub-programs depending on whether SIPs or DIPs are utilized, namely, STSR program and DTSR program. STSR stands for SIP-based Test Suite Reduction, which reduces the number of test cases according to the static interaction patterns. As for DTSR, it stands for DIP-based Test Suite Reduction, which reduces the number of test cases by employing dynamic interaction patterns. Figure 4.1 and Figure 4.2 show the structure of the STSR and DTSR program, respectively.

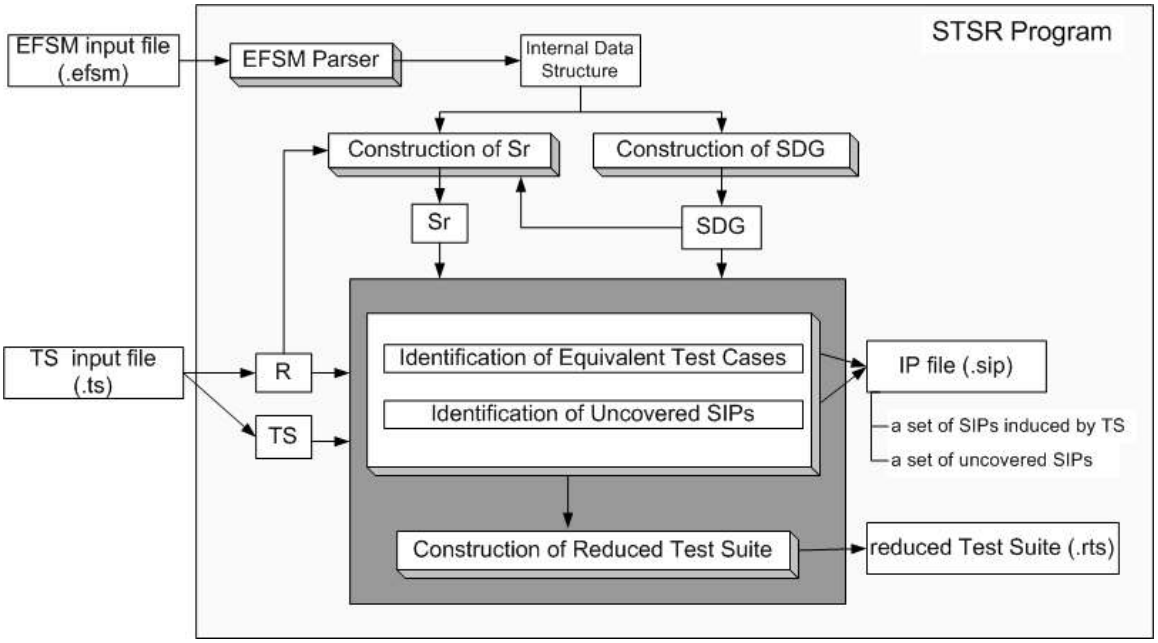

Figure 4.1 Structure of STSR

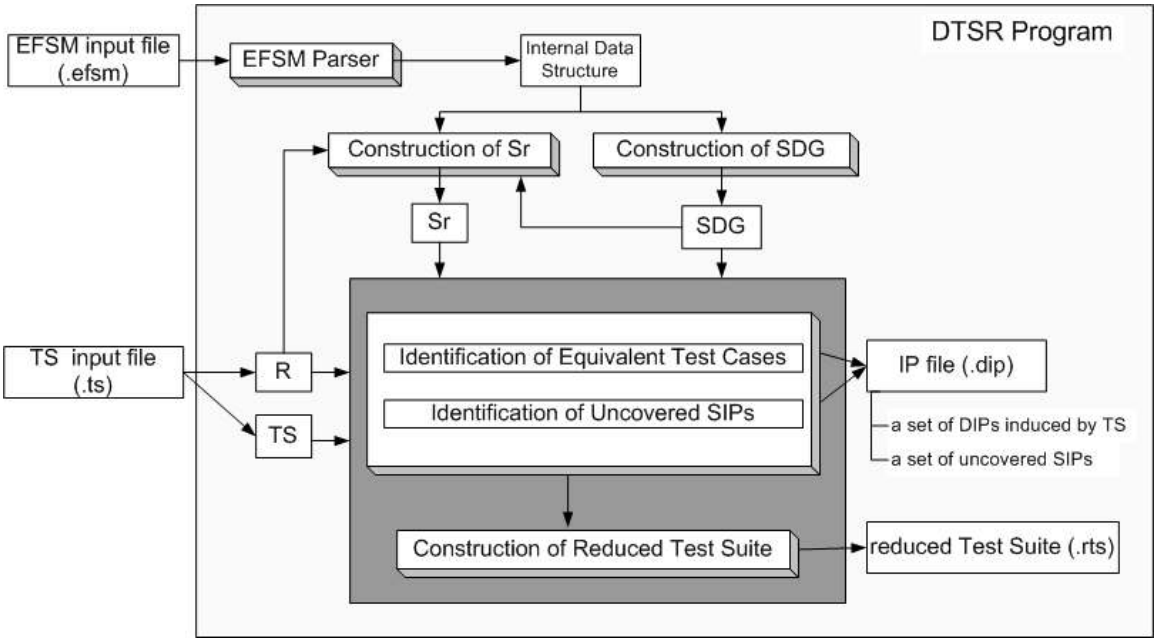

Figure 4.2 Structure of DTSR

Basically, STSR and DTSR have the same architecture and performs the following tasks:

- Phase 1: Construction of SDG;
- Phase 2: Identification of Equivalent Test Cases and Uncovered SIPs;
- Phase 3: Construction of Reduced Test Suite;

It must be noted that Phase 1 was implemented by ASERT Lab members: Tuong Nguyen and Yan Gao where the EFSM parser developed by Tuong yields internal data structures which are subsequently used to build an SDG. SDG construction was developed by Gao based on the algorithm for generating the SDG from an EFSM in [37].

The implementations of these three phases are discussed in detail in Sections 4.4 and 4.5, respectively for STSR and DTSR. Section 4.2 gives the input file formats of STSR and DTSR, while Section 4.3 gives the output file formats.

4.2 Input File Formats

As shown in figures 4.1 and 4.2, STSR and DTSR require two input files: EFSM input file and TS input file. The EFSM input file is a file representing an EFSM. In this thesis, such EFSM files are given the ".efsm" extensions, and are defined formally below using the Backus-Naur Form (BNF):

Table 4.1 BNF definition of an EFSM input file

<efsm> ::=
efsmId
numStates startStateIndex exitStateIndex
<transitions>
<transitions> ::=

```
        <transition> | <transitions> <transition>
<transition> ::=
        transition transitionId
        sourceStateIndex destinationStateIndex
        <requirement>
<requirement> ::=
        [<input>]
        [<enablingPredicate>]
        /
        [<actions>]
<actions> ::=
        <action> | <actions> <action>
<action> ::=
        <output> | <assignment> | <set> | <reset> | <procedureCall>
<input> ::=
        inputId ( [<parameters>] )
<output> ::=
        outputId ( [<parameters>] )
<enablingPredicate> ::=
        <variableIds> [/* booleanExpression */]
<assignment> ::=
        <variableId> := <expression>
<set> ::=
```

```
        set ( constant , timerId )
<reset> ::=
        reset ( timerId )
procedureCall ::=
        procedure ( procedureId ( <variableIds> [; <variableIds>]) ) { <pbrDefs> }
<parameters> ::=
        <parameter> {, <parameter>}*
<parameter> ::=
        <variableId> | constant
<variableIds> ::=
        <variableId> {, <variableId>}*
<pbrDefs> ::=
        <pbrDef> | <pbrDefs> <pbrDef>
<pbrDef> ::=
        <variableId> := <expression> ;
<expression> ::=
        function ( <variableIds> ) | constant
<variableId> ::= id
```

An example ".efsm" file for the EFSM of the ATM system in Figure 2.3 is given in Appendix A.7.

Another input file, TS input file, is a file representing a test suite. A TS file consists of a set of requirements where each requirement is represented by a transition under test (tut) and a collection of test cases. A test case is a complete sequence of transitions that

starts at the entry node and ends at the exit node of the EFSM. TS input file have been given the ".ts" extension. The detailed definition of TS files in BNF is presented below:

Table 4.2 BNF definition of a TS input file

```
<ts> ::=
        efsmId <tuts> <tests>
<tuts> ::=
        <tut> | <tuts> <tut>
<tut> ::=
        transitionId
<tests> ::=
        <test> | <tests> <test>
<test> ::=
        test testId <transitionSeq>
<transitionSeq> ::=
        transitionId | <transitionSeq> transitionId
```

An example TS file for the EFSM of the ATM system presented in Figure 2.3 is given in Appendix A.8.

4.3 Output File Formats

STSR and DTSR each generates two output files: a reduced TS file and an IP file. The reduced TS file is a file representing the reduced test suite where redundant test cases have been eliminated. A reduced TS file is given the ".rts" extension and shares the

format of original TS file. The detailed definition of reduced TS files in BNF is presented below:

Table 4.3 BNF definition of a reduced TS input file

```
<ts> ::=
        efsmId <tests>
<tests> ::=
        <test> | <tests> <test>
<test> ::=
        test testId <transitionSeq>
<transitionSeq> ::=
        transitionId | <transitionSeq> transitionId
```

Another output file, IP, represents, for each TUT, two sets of interaction patterns. The first set consists of unique interaction patterns induced by test cases in a TS input file, which are either static or dynamic interaction patterns w.r.t. the TUT. Each interaction pattern comes with a group of equivalent test cases that exhibit it. Each test case is referred to in the file by its test id. The second set consists of static interaction patterns w.r.t. the TUT that are not covered by any test case in the original TS file. The extension of the file depends on the program used. STSR generates a ".sip" file, while DTSR generates a ".dip" file. The IP file is defined using the Backus-Naur Form (BNF) below:

Table 4.4 BNF definition of an IP output file

```
<sip/dip> ::=
        efsmId <tut> <ips>
```

<tut> ::=

transitionId

<ips> ::=

<ip> | <ips> <ip>

<ip> ::=

ip ipId [<testIds>] <nodes>

<testIds> ::=

testId | <testIds> testId

<nodes> ::=

<node> | <nodes> <node>

<node> ::=

node nodeIndex <label> [<adjacencySet>]

<label> ::=

transitionId

<adjacencySet> ::=

<reverseSet> | <nonreverseSet>

<reverseSet> ::=

<reverse> | <reverseSet> <reverse>

<reverse> ::=

inc sourceIndex <dependencyType>

<nonreverseSet> ::=

<nonreverse> | <nonreverseSet> <nonreverse>

<nonreverse> ::=

```
        out destinationIndex <dependencyType>
<dependencyTye> ::=
        dat | ctl
```

It is noted that an IP output file distinguishes a type of interaction pattern according to the prefix of ipID i.e., "S", "D" and "U" denote static, dynamic and uncovered static interaction pattern, respectively. Examples of ".sip" files for the EFSM of the ATM system given in Figure 2.3 can be found in Appendix A.9.

4.4 STSR Program

STSR uses static dependency analysis to reduce the size of test suites without significantly reducing their fault-detection capability. It eliminates repetitive test cases, i.e. all test cases except one that exhibit the same static interaction pattern. As mentioned earlier, STSR can be broken down into three phases.

Phase 1: Construction of SDG & S_r

Given the two input files, STSR checks that the test suite belongs to the EFSM system (e.g., by checking if efsm ids in EFSM and TS files are identical). Then, the EFSM file is analyzed by lexical and parser utilities of the Unix environment, Lex and Yacc, to get the EFSM information, classify the variables and their occurrences in each transition, and form the EFSM internal data structure. Lex generates lexical analyzers. The code generated by Lex is used to read input characters and produce a sequence of tokens that a parser can use for syntax analysis. Yacc generates parsers that get the tokens and fill the internal data structure of the program according to the given grammar. Based on this internal data structure: control and data dependencies are identified and captured

to build the SDG. For example, from the EFSM input file given in Appendix A.7, the fragment of internal data structure of the SDG representing the fact that there is a data dependency from transition T1 to transition T2 is represented here:

Table 4.5 An Example of Data Dependency Representation

EFSM id: ATM_System
Static Dependency Graph (SDG):
Source node index: 0, Label: T1
Destination node index: 1, Label: T2
Number of edges: 3
List of edges: DType(Data=0,Control=1,CUse=2,PUse=3)
(Variable, DType, OOrder of def, OOrder of use)
(pin,0,2,3)
(attempts,0,3,2)
(attempts,0,3,5)

To generate the SDG from a given EFSM efficiently, the algorithm proposed in [37] is used. The complexity of SDG generation is quadratic in S which is a set of states in an EFSM [37].

In this phase, besides the construction of SDG, a set S_r of all possible SIPs w.r.t. each requirement under test r is constructed. In order to construct S_r, the information from SDG, the EFSM internal data structure and a set of requirements are needed. At this point, only the set of requirements has not yet been obtained; hence, a TS input file is now required. As stated previously, a TS input file consists of a set of transitions under test and a collection of test cases where each test case is a sequence of transitions starting at the entry node and ending at the exit node of the corresponding EFSM. At this moment, those sets in the TS input file are extracted to form R and TS which stand for a set of requirements under test and a set of test cases, respectively. The algorithm to generate S_r proposed in [37] is used to obtain S_r in this phase.

Phase 2: Identification of Equivalent Test Cases and Uncovered SIPs

In this research, we assume that an individual requirement can be mapped into one transition in an EFSM so that *requirement under test* r and *transition under test* TUT can be referred interchangeably.

In this phase, an IP output file, ".sip" is constructed from R and TS (obtained from the previous phase). As mentioned earlier, ".sip" is composed of, for each requirement under test r, two sets of interaction patterns: a set S'_r of SIPs w.r.t. r induced by test cases in TS and a set of SIPs that are not covered by any test case in TS. The motivation to report uncovered SIPs is based upon the fact that some static patterns of interactions of the system under test may remain uncovered by the TS input file. As a consequence, the reduced test suite obtained in Phase 3 may not be capable of detecting some faults w.r.t. a TUT. Hence, STSR reports a set of uncovered SIPs w.r.t the TUT in ".sip".

In order to generate a ".sip" file, STSR, for each $r \in R$, constructs a set of SIPs w.r.t. r induced by test cases in $TS_r \subset TS$ and a set of uncovered SIPs w.r.t. r as follows:

- Step 1: STSR identifies a set of equivalence classes of transition sequences having the same SIP w.r.t. the TUT, which is subsequently used in Phase 3 to construct a reduced TS file.
- Step 2: STSR identifies and reports SIPs w.r.t. TUT that are not covered by any test case in TS.

Details of the execution of these two steps are given below:

For each transition under test TUT,

/* *Step1*/

1. STSR forms a set of test cases (TS_r) w.r.t. TUT by extracting those test cases from the TS file in which the TUT occurs at least once.
2. STSR identifies sets of equivalent test cases in TS_r by investigating transition sequences that exhibit the same static interaction pattern (SIP). Thus, let $TS_r(SIP_i)$ be the equivalence class of transition sequences having the same SIP_i. For each transition sequence $ts \in TS_r$,
   - STSR forms SIP_{ts} from ts using SDG obtained in Phase 1. The algorithm for generating the SIP_{ts} from SDG is presented in Section 3.2.1.
   - STSR puts ts in $TS_r(SIP_i)$ where SIP_i is equivalent to SIP_{ts}. The algorithm for comparing two SIPs is presented in Section 3.2.2.
   - STSR puts SIP_i in the set S'_r of static interaction patterns for r induced by TS_r.

3. STSR exports all $SIP_r \in S'_r$ (where r corresponds to the TUT) as well as a group of test cases (referred to by test id) from $TS_r(SIP_r)$ into ".sip".

/* *Step 2* */

4. STSR inputs a set of all possible SIPs w.r.t. the TUT, S_r.
5. STSR identifies which $SIP_r \in S_r$ (if any) is not covered by any $ts \in TS_r$ using S_r and S'_r (obtained in Step 1). The algorithm to identify uncovered SIP_r is presented in Section 3.2.3.
6. STSR reports $S_r - S'_r$ by appending such information to ".sip".

Phase 3: Construction of Reduced Test Suite

In the previous phase, repetitive test cases have already been identified in terms of equivalent test cases that exhibit the same SIP w.r.t. the TUT. Therefore, reduced test suite can be constructed by simply selecting randomly one test case of each equivalence class of transition sequences having the same SIP w.r.t. the TUT. More formally,

let RTS = {ts | ts is a complete sequence of transitions};

RTS ← ∅

for each $r \in R$ in the TS file

$RTS_r \leftarrow \varnothing$

for $i = 1$ to $|S'_r|$ do

select randomly one ts from $TS_r(SIP_i)$ and $RTS_r \leftarrow RTS_r \cup \{ts\}$

end for

$RTS \leftarrow RTS \cup RTS_r$

end for

At this point, STSR has formed the reduced test suite and exported the contents of the TS file to ".rts" file (including efsm id and set of transitions under test) where the repetitive test cases have been eliminated.

4.5 DTSR Program

DTSR uses dynamic dependency analysis to reduce the size of a given test suite, without significantly reducing its fault-detection capability. DTSR implementation is largely similar to that of STSR, except for some specific details in phases 2 and 3. Those differences are described here.

Phase 2: Identification of Equivalent Test Cases and Uncovered SIPs

Similar to STSR, DTSR extracts *R* and TS from the TS input file in Phase 1. Again in this phase, an IP output file is constructed from *R* and TS; however, the extension of the constructed IP file is ".dip" as opposed to ".sip" in STSR.

The difference between STSR and DTSR in this phase stems from the fact that in DTSR, dynamic interaction patterns are used to define equivalence classes of transition sequences. Therefore, ".dip" is composed of, for each requirement under test *r*, two sets of interaction patterns: a set D_r of DIPs w.r.t. *r* induced by test cases in TS and a set of SIPs that are not covered by any test case in TS. The motivation of DTSR to report uncovered SIPs is based upon two facts: first, the objective of testing (as mentioned in Chapter 3) is to cover different patterns of interactions w.r.t. each TUT and second, the number of SIPs w.r.t. each TUT is bounded. Therefore, a complete test suite should at least cover all static interaction patterns w.r.t. each TUT. Thus, DTSR reports uncovered SIPs w.r.t. each TUT in ".dip".

DTSR generates ".dip" using the same principle as is used in STSR to generate ".sip". That is, DTSR, for each $r \in R$, constructs a set of DIPs w.r.t. r induced by test cases in $TS_r \subset TS$ and a set of uncovered SIPs w.r.t. r as follows:

- Step 1: DTSR identifies a set of equivalence classes of transition sequences having the same DIP w.r.t. the TUT, which is subsequently used in Phase 3 to construct a reduced TS file.
- Step 2: DTSR identifies and reports SIPs w.r.t. TUT that are not covered by any test case in TS.

Details of the execution of these two steps are given below:

For each transition under test TUT

/* *Step 1* */

1. DTSR forms a set of test cases (TS_r) w.r.t. TUT by extracting those test cases from TS in which the TUT occurs at least once.
2. DTSR identifies sets of equivalent test cases in TS_r by investigating transition sequences that exhibit the same dynamic interaction pattern (DIP). Thus, let $TS_r(DIP_i)$ be the equivalence class of transition sequences having the same DIP_i.

   For each transition sequence $ts \in TS_r$,

   - DTSR forms DDG_{ts} from *ts* using the algorithm for generating the DDG_{ts} for a given *ts* which is presented in Section 3.3.1
   - DTSR generates DIP_{ts} from DDG_{ts} using the algorithm for generating DIP_{ts} from DDG_{ts} which is presented in Section 3.3.2.

- DTSR puts *ts* in $TS_r(DIP_i)$ where DIP_i is equivalent to DIP_{ts}. The algorithm for comparing two DIPs is presented in Section 3.3.3.
- DTSR puts DIP_i in the set D_r of dynamic interaction patterns for *r* induced by TS_r.

3. DTSR exports all $DIP_r \in D_r$ (where *r* corresponds to the TUT) and, associated with it, a group of test cases (referred to by test id) from $TS_r(DIP_r)$ into ".dip".

/* *Step 2* */

4. DTSR inputs a set of all possible SIPs w.r.t. TUT, S_r.
5. DTSR identifies which $SIP_r \in S_r$ (if any) are not covered by any $ts \in TS_r$ using S_r and D_r (obtained in Step1). This is based on the fact that for each transition sequence *ts*, DTSR maps DIP_{ts} into a SIP_{ts} by collapsing group of nodes representing the same transition into a single node.
6. DTSR reports S_r - D_r by appending such information to ".dip".

Phase 3: Construction of Reduced Test Suite

Just like STSR, DTSR randomly selects one test case from each equivalence class of transition sequences having the same DIP w.r.t. the TUT. More formally,

let RTS = {ts | ts is a complete sequence of transitions},

RTS ← ∅

for each $r \in R$ in the TS file

$RTS_r \leftarrow \varnothing$

for $i = 1$ to $|D_r|$ do

select randomly one *ts* from $TS_r(DIP_i)$ and $RTS_r \leftarrow RTS_r \cup \{ts\}$

end for

$RTS \leftarrow RTS \cup RTS_r$

end for

DTSR thus generates the reduced test suite and places it in the output ".rts" file. This file shares the format of the input TS file, except for repetitive test cases which have been removed.

In the next chapter, we evaluate our implementation through four case studies. The results of these experiments are also presented and analyzed.