Automatic Protocol Test Suite Derivation

Samuel T. Chanson¹ and Jinsong Zhu
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada V6T 1Z2

Abstract

In this paper, an automatic protocol test case generator which generates both test sequence and test data is presented. The underlying model is an extended finite state machine (EFSM) and Estelle is used as the formal description language. First, test sequences are generated using a combined control and data flow testing method. A set of path conditions associated with each test sequence is obtained using symbolic execution techniques. By solving the path conditions as a group of constraints, test data are then automatically generated. A prototype system implementing the above procedures has been built. The system has been used to generate test cases for real protocols such as TP0, FDDI and LAPB, and has been integrated in the UBC protocol testing environment.

1 Introduction

There is considerable interest in protocol testing in recent years, both in theory and practice. Since testing is a costly process, much effort has been expended on minimizing the length of test sequences and automating the process as much as possible. Automatic test sequence generation also enhances correctness and completeness by eliminating human errors. In general, two aspects of the protocol have to be considered for testing purposes: control flow and data flow. Control flow is often modelled using a finite state machine (FSM) while data flow is analyzed with the help of static analysis techniques [10, 12]. These two aspects have largely been studied independently. In [2], we proposed a combined approach which integrates control and data flow testing into a unified framework. By doing so, the total length of test cases is reduced without sacrificing fault coverage. The algorithms that are used to automate the approach were also presented.

After a test sequence is generated, test data have to be carefully selected to enable the test sequence to be executed as well as to observe the data flow effects. Since a test sequence can have constraints of arbitrary complexity, automatic data selection is difficult and has traditionally been done by hand. The constraint satisfaction problem is also known to be NP-complete for even three binary variables [8]. Nevertheless, it is desirable to be able to generate the data values needed for testing automatically.

Our solution is based on the observation that communication protocols tend to have simple predicates which usually only relate to the current input event for a given transition. To obtain the

¹ S.T. Chanson is with the Hong Kong University of Science & Technology.

constraints associated with a test sequence, a set of path conditions in terms of input parameters are computed using symbolic evaluation techniques [3]. A heuristic constraint-solving procedure is presented that works efficiently and effectively for simple predicates but does not give up on complex ones. The test data are automatically selected by solving the constraints for the constrained parameters and randomly choosing values for the unconstrained ones.

The schematic diagram of the automatic test generation (ATG) system is shown in Figure 1. The tool set takes an Estelle specification of the protocol to be tested, translates it into an internal representation and generates a set of test sequences as well as the path constraints for each test sequence. The constraint solver is then invoked to produce test data that allow the sequences to be executable. In order to increase the portability of the output, the test sequence generated is translated into TTCN behavior descriptions and the test data into TTCN constraint declarations.

Figure 1: Automatic Protocol Test Generator Architecture

The remainder of the paper is organized as follows. Section 2 gives a short review of the test sequence generation approach. The next section describes techniques used in producing path conditions (or constraints) for each test sequence. Section 4 focuses on test data generation based on the constraint solver. Section 5 deals with the constraint matching problem for a receive event. Section 6 illustrates with an example our experience in applying the tool to some real life protocols. Finally Section 7 concludes the paper.

2 Test Sequence Generation

In [2], we proposed a unified approach to protocol test sequence generation based on the Extended Finite State Machine (EFSM) model. Unlike the Finite State Machine Model (FSM), EFSM contains both control and data information. Until recently, control and data flows are treated separately. Coverage could be compromised and the overall length of the test cases is longer because of separate tests. Our approach combines the two parts into one framework by generating a test sequence capable of testing both parts simultaneously, while maintaining the fault coverage for each part. The length of the test sequence is also restricted within a reasonable bound.

For control flow testing, we proposed to use a cyclic characterizing sequence (CCS) as the state identification sequence of a state. A CCS is composed of the usual state identification sequence (such as UIO, which is what we used in our tool) plus a transition sequence from the end state of the identification sequence back to its original start state, i.e., the state it is trying to identify. CCS has the following properties:

- (1) If s_p is the tail state of a sequence

$$SEQ_{0,p} = \{(s_0, s_1; i_0/o_0), (s_1, s_2; i_1/o_1), \dots, (s_{p-1}, s_p; i_{p-1}/o_{p-1})\},$$

and no $s_i, i = 0, \dots, p$ is a converging state, then $SEQ_{i,p} @ CCS(s_p)$ is a characterizing sequence for all states $s_i, i = 0, \dots, p$.

This property ensures that all states along SEQ are tested by a single test sequence $SEQ_{0,p}@CCS(s_p)$. It is equivalent to using overlaps between test subsequences.

(2) If along $SEQ_{0,p}$ there are converging states $s_{k_1}, s_{k_2}, \dots, s_{k_j}$, then after inserting a CCS for each converging state to form a new sequence

$SEQ'_{0,p} = \{(s_0, s_1; i_0/o_0), \dots, (s_{k_j}, s_{k_{j+1}}; i_k/o_k), CCS(s_{k_{j+1}}), \dots, (s_{p-1}, s_p; i_{p-1}/o_{p-1})\}$,
 $SEQ'_{i,p}@CCS(s_p)$ is a characterizing sequence for all states $s_i, i = 0, \dots, p$.

This is because after a CCS is inserted for a converging state, the state can then be uniquely identified instead of being confused with its converging companions (i.e., the other converging states with outgoing transitions that lead to the same state).

With these two properties, test sequences can be generated in a straightforward manner. First, generate a set of linearly independent circuits each of which starts and ends at the initial state. Then for each circuit, append the initial state's CCS and insert a CCS for each converging state if any. This set of circuits¹ constitutes the test suite for control flow testing. It tests all transitions and also identifies all states. Therefore it can detect all output faults as well as transfer faults. The requirement of linear independency among the circuits limits the number of circuits to $e - n + 1$, where e and n are the number of edges and vertices respectively [2], thereby restricting the length of the test suite to the upper bound given by $e(e - n + 1)$.

The reason to devise this new scheme for FSM (i.e., control flow) test sequence generation is that we found the existing methods are not suitable for integrating with data flow testing.

In data flow analysis, control dependency and data dependency are defined among transitions. If there is a variable defined in a transition and used by another transition, then the latter variable is said to be control or data dependent upon the former, depending on whether the variable is used in transition predicates (*p-use*) or the program segment (*c-use*). Note that there must be at least one def-clear path between a dependency pair, which means the associated variable should not be redefined along the path. By scanning all variables defined in a transition and searching for their global uses in other transitions, the dependency relations can be derived for all def-use pairs. The all-du-paths criterion stipulates that all def-clear paths between a def-use pair be exercised at least once during testing, making it a stringent criterion only inferior to the all-paths criterion. This may seem rather expensive at first glance, but experience shows that it is not so in most cases, since many du-paths are interleaved and thus can be merged and tested at the same time.

We first generate all dependency pairs, and then each one is connected with a preamble leading to its start transition, a postamble from the end of the pair to the initial state s_0 , and a CCS of s_0 . This generates a set of circuits that covers all du-paths and, at the same time, satisfies the requirements of control flow testing. Our earlier claim about the method's coverage can then be justified. A detailed description of the algorithms can be found in [2].

3 I/O Constraints Generation

After the test suite is generated, test data must be chosen to exercise each test case in the test suite. As we know, a transition may have predicates associated with it. Predicates are conditions which must be satisfied in order to fire the transition. If a predicate contains input parameters only, the value of the parameters can be selected directly from their domains based on the conditions expressed by the predicate. When a predicate contains context variables, since they cannot be controlled and observed from outside of an implementation under test (IUT), it is necessary to

¹Each circuit corresponds to a test case. We shall use these two terms interchangeably in the paper.

trace backwards along the circuit until their definitions by constants or some input parameters are found. This is called *predicate interpretation*. After interpretation, predicates are represented only by constants and/or input parameters, and are controllable from outside. They are also referred to as *input constraints*.

There may also be constraints on output parameters, which specify what the output parameters should be after the execution. Similar to predicate interpretation, an output parameter also needs to be interpreted and resolved to constants and/or input parameters so that the data flow effects are observable. We shall call them *output constraints*. To derive these I/O constraints, path-dependent symbolic evaluation techniques [3] may be used for interpretation.

There are two kinds of path-dependent symbolic evaluation techniques, namely, forward expansion or backward substitution [3]. While forward expansion techniques begin at the start transition and develops the symbolic representations as each predicate in the circuit is interpreted, backward substitution works from the final transition toward the start transition. Since a predicate or an output parameter needs to be interpreted only when it contains context variables, backward substitution is used in our constraint generation procedures as explained below.

Input Constraint Generation

Since each context variable used in a predicate must have been defined in some previous transition along the circuit, we can generate an interpretation of the predicate by traversing the circuit in a backward fashion using the backward substitution technique. In the likely case that not all transitions have predicates, or not all predicates contain context variables, this will save a lot of computation. Another advantage of this method is that no storage is required for saving the symbolic values of the variables. The following Algorithm-IC implements this method and generates a set of equality and inequality constraints on the input parameters for each transition. These constraints define a subset of the input space which allows the execution of the associated test case.

Algorithm-IC: Input Constraint Generation for a Circuit

Input: A circuit containing transitions $t_0, t_1, \dots, t_n$

Output: A set of input constraints for the circuit

Procedure 1: A recursive procedure doing backward substitution

```

/* x: a context variable, i: transition index */
procedure bksub( $x, i$ );
begin
  if  $x$  is defined in  $t_i$  then begin
    substitute  $x$  with its definition  $D$ ;
    for all context variables  $y \in D$  do
      bksub( $y, i$ );
    end
  else
    bksub( $x, i - 1$ );
  end

```

Procedure 2: Input constraints generation

```

for  $i = n$  downto 1 do
  for each variable  $x \in \text{Predicate}(t_i)$  do
    bksub( $x, i$ );

```

Procedure 1 in Algorithm-IC is actually a depth-first search until the definitions of all use variables in a predicate is found. This procedure is recursive since a variable used in a definition can be further defined in terms of other variables.

Note that different paths may interpret the same predicate differently. Therefore a predicate interpretation is restricted to the test case concerned. The interpretation together with the type constraints of the input parameters determines an allowable input domain for the test case. Test data must then be chosen from this domain. Figure 2 shows a test case taken from TP0 with the following input constraints:

t_0 : TCREQ.qtsReq ≥ 0
 t_1 : DR.disReason = 0
 t_2 : 300 $\neq 0$ (always true, omissible)
 t_3 : TCRES.qtsReq ≤ 26

For example, in t_3 , $qtsEst$ is a context variable used in its predicate. By backward substitution, it is interpreted to the constant 26 defined in t_2 .

Figure 2: Constraint generation example

Output Constraint Generation

While the input constraints specify the domains from which input data should be selected for a test case to be executable, the output constraints are mapping functions from the input domain to the output domain. If an output parameter is assigned by a local input parameter, its constraint can be directly observed in a single test step (i.e., one transition); no global data flow analysis is involved. Otherwise, the output parameter must be substituted by constants or input parameters upstream along the circuit in much the same way that input constraints are generated.

In general, suppose we have a test case with n steps:

$$i_0/o_0, i_2/o_2, \dots, i_{n-1}/o_{n-1}.$$

For each i/o pair i_k/o_k , their parameters are:

$$i_k = i_k(p_{k0}, p_{k1}, \dots), o_k = o_k(q_{k0}, q_{k1}, \dots).$$

The first step is assumed to start from the initial state, so the values of context variables are known. After substitution, each parameter in an output $o_k, k = 0, \dots, n-1$, will be, in general, a function of parameters in i_0 up to i_k (some terms may be missing), i.e.,

$$q_{kj} = F(p_{00}, p_{01}, \dots, p_{10}, p_{11}, \dots, p_{k0}, p_{k1}, \dots).$$

Similar procedure is applied to predicate interpretation.

For example, in Figure 2, the output constraints for each step are as follows:

```

t0: CR.srcRef = 300
      CR.option = 100
      CR.cgAddr = t0:TCREQ.fromAddr
      CR.cdAddr = t0:TCREQ.toAddr
      CR.maxTpdu = 80
t1: NDREQ.disReason = t1:DR.disReason
      TDIND.disReason = t1:DR.disReason
      TDIND.userReason = t1:DR.addClrReason
t2: TCIND.toAddr = t2:CR.cdAddr
      TCIND.fromAddr = t2:CR.cgAddr
      TCIND.qtsPro = 26
t3: DR.dstRef = t2:CR.srcRef
      DR.disReason = 0
      DR.addClrReason = 0
      TDIND.disReason = 0

```

Note that input parameters are referenced by their corresponding transitions.

4 Constraint Solver

To generate test data, the input constraints have to be solved. As mentioned before, the general constraint satisfaction problem is NP-complete. In [7], only integer and boolean parameter types are considered, and operators are restricted to “+” and “−”. Integer linear programming techniques are employed to find solutions for the constraints. The restrictions are acceptable in most cases. However, the computational cost is expensive and the scheme will fail in the cases that do not satisfy the requirements. In [5], heuristics were used to solve mutant-killing constraints. The strategy was efficient for constraint systems that contain clauses with a simple form. We select the heuristic approach also since protocol specifications typically have simple predicates. Our heuristic procedure is similar to the ideas in [5] but also contains features that take advantage of the special characteristics of protocol processing. It works efficiently for simple predicates. For complex constraints, the procedure takes longer time but does not give up.

To quantify our belief that protocols have simple predicates we have examined some Estelle specifications, most of which were obtained from the University of Delaware which started to collect Estelle specifications of protocols a few years ago. Specifications inspected include Abracadabra, Alternating Bit Protocol, DQDB, Connectionless Network Protocol, Sliding Window, ISO Transport, ISO Session, ISO Virtual Terminal Protocol, etc. Among the 197 predicates examined, 13 have no variables after interpretation: the value of the expression can be evaluated immediately.

162 of them have one variable, which account for 90.5% of the total predicates. Predicates that have more than one variables make up only 2.1% altogether. Moreover, the number of terms in a predicate is one in most cases (66%), and most variables are of the integer, boolean, or enumeration type (in fact, we didn't see any real type variables). We also noted that in most predicates (about 98%), the terms (one relational constraint) are mostly independent, i.e., they seldom have common variables such as, for example, x in $(x + y > 5) \wedge (x + z < 10)$. This means that most predicates contain separate, unrelated terms. Although the number of Estelle specifications examined was rather limited due to their availability, the predicates studied reflect commonly used structures in protocol specifications. The above results strongly suggest that efficient and effective heuristics exist in solving the constraints in most cases.

Consider a test case (i.e., a circuit) with transitions $T_1, T_2, \dots, T_n$. Each transition T_k may be associated with a corresponding input event I_k , and a constraint C_k (both could be empty). After predicate interpretation along this circuit, C_k (if not empty) will contain only constants and input parameters in I_1 up to I_k . The non-empty constraints form a constraint system for the test case. It is obvious that for any two constraints C_i and C_j ($i < j$) in the constraint system, parameters in C_i may appear in C_j but not the other way around. This implies that if we select test data progressively from C_1 to C_n , the values of the new parameters in C_k need only be substituted in the constraints that follow, i.e., $C_j, j > k$. This is like we had sorted the constraints to eliminate backward substitution to improve efficiency.

Algorithm-DR: Domain reduction for input parameters

Input: A constraint system C and initial variable domains D_x for x

Output: Reduced domains for all variables

Initially, D_x contains a list of lower and upper bounds.

1. **for** each constraint p in C of the form xRc **do begin**
2. find the interval (a, b) that c falls in;
3. **if** $R = ">="$ **then** $D_x = (c : b, \dots)$;
4. **if** $R = ">"$ **then** $D_x = (c + 1 : b, \dots)$;
5. **if** $R = "<="$ **then** $D_x = (\dots, a : c)$;
6. **if** $R = "<"$ **then** $D_x = (\dots, a : c - 1)$;
7. **if** $R = "="$ **then** $D_x = (c : c)$;
8. **if** $R = "\neq"$ **then** $D_x = (\dots, a : c - 1, c + 1 : b, \dots)$;
9. **if** D_x contains any contradictory interval **then**
10. **return** " C not satisfiable";
11. remove p from C ;
12. **end**

In solving constraint satisfaction problems, a strategy that is often used before beginning to search for a solution is domain reduction. The search space could be dramatically pruned in some cases (though we should not expect it would always be very effective). Algorithm-DR shows our proposed domain reduction procedure. Since single variable constraints are the most frequent, the procedure is performed for constraints of the form xRc , where x is an input parameter, c is a constant, and R a relational operator. Initially, a variable's domain is an array of lower and upper bounds (for enumeration type, a value can be considered as having the same lower and upper

bound), which should be determined by the user ². The lower and upper bounds are modified according to the single variable constraints for x . When R is “>”, “<”, or “≠”, we use 1 as the smallest granularity for x , implicitly assuming x to be an integer (in fact, real numbers are rarely used in protocol specifications). Other values can be used as the granularity for other data types. If R is “=”, then c is assigned to x , and all subsequent occurrences of x are replaced by c . This could produce new single variable constraints and the process is iterated until no further reduction is possible. During reduction, if a contradictory condition is obtained, say the upper bound is less than the lower bound, then the procedure aborts and a message that the constraint system C cannot be satisfied (or insolvable) is returned. This helps to identify the inexecutable test cases at an early stage. When no further reduction can be done, Algorithm-CS is employed to find a solution for the constraint system. The procedure repeatedly assigns values to each variable until a solution is found. The assignment order for the variables is based on the following heuristics:

(1) The variables occurring in the most number of constraints are assigned value first. This is because these variables maximally constrain the rest of the search space.

(2) If there are more than one variable satisfying condition (1), the variable having the smallest domain size is assigned value first. This heuristic was also used in [5]. It is based on the intuition that variables with larger domain size would have more chances to take on a value that satisfies the constraint system.

Algorithm-CS: Constraint Solver

Input: A constraint system C and variable domains D_x for x

Output: Variable values satisfying C or C is not satisfiable

Step 1: Convert each constraint to a disjunctive normal form, which consists of a disjunction of relational constraints;

Step 2: Compute for each variable first occurring in some C_i the number of its occurrence in subsequent constraints, denoted as N_x for x ;

Step 3: Start from C_1 , generate values for all variables in the constraints, using the following iteration procedure.

Iteration: For C_i ,

- 1) Choose the variable x with the largest N_x . If there are more than one such variables, choose the one with the smallest domain size;
- 2) Randomly choose a value k for x from D_x ;
- 3) Substitute all occurrences of x in subsequent constraints with k . If the substitution produces single variable equations, solve them and get the value for the single variable.
- 4) If all occurrences of x is successfully substituted (ie, no dead end was encountered), repeat 1)-3) for other variables in C_i ; otherwise, choose another value for x . If the number of trails exceeds some predetermined bound, C is declared as insolvable.

²In some cases, variable domains can be automatically obtained from the formal specification, but often, a domain can be reduced considerably for a given implementation under test. The refinement can help to increase the efficiency of the constraint solver.

When assigning a value to a variable, the value is randomly chosen from the variable's domain. When dead end (i.e., a given value of the variable x cannot be used to satisfy all occurrences of x) is encountered, other values (if any) for the variable are tried. This procedure is repeated until all constraints are satisfied or a maximum number of trials is performed. In the latter case, the constraint system is assumed to be unsolvable.

Now, it is possible that a dead end was reached because of the choice of some parameter value. If the constraint system consists of a small number of variables, it may be possible to rerun the algorithm with different random choices and find a solution. This simple strategy may not perform as well as backjumping [4]. However, it is very difficult to determine which previous variable was the direct reason for the dead end (for backjumping), and the possible infinite domains prevent an exhaustive search for all values. Our experience with a number of protocols showed that the constraint system for the test cases could be solved with no or a small number of rerunnings of the procedure.

5 Receive Constraint Matching

Once the values of all parameters in an event are selected (by means of the constraint solver for constrained parameters, or random selection for unconstrained ones), the parameters and their values are represented using the TTCN notation which is further translated into an internal tree form (called value E-node [1]) by a tool. From the viewpoint of the tester, an input event (in the *when* clause) corresponds to a send event (ASP/PDU) to the IUT, while an output event corresponds to a receive event from the IUT. Specific values must be provided for a send event to encode it. Routines that perform the encoding from a value E-node to BER (Basic Encoding Rules) octet stream or to a particular PDU encoding (mostly for layers below the application layer) are provided. For a receive event, the received encoded event is first converted back to its value E-node form and compared against the expected constraint for that event. When the expected constraint contains specific values for all fields, it is sufficient to perform a binary comparison with the received one to verify its correctness; no conversion to the value E-node is necessary in this case. However, sometimes in the expected constraint, a field (any value or in the value list) could be omitted. In such cases, a simple binary comparison will not work.

In TTCN [9], a set of matching mechanisms is defined: specific values (which is the basic mechanism), complement, omit, any value, any or omit, value list, range, super set, subset, and permutation. These new data types are extensions of ASN.1 in TTCN. Accordingly, we have extended the E-node, an internal representation of ASN.1 [11, 1], to allow for the specification of these data types.

The value E-nodes are organized as a tree. In each E-node, there is a flag indicating the type of constraint, which can be a specific value, omit ("—"), any ("?"), value list, and integer range. When the constraint is not a specific value, the E-node will contain fields which supplement the constraint, such as a pointer to a value list, and range for an integer. A tree attachment mechanism is also provided to handle structured definition of a constraint. Figure 3 shows an example of a value E-node tree corresponding to the TTCN specification in Table 1 and 2.

The value E-node for a receive constraint is saved on disk as a file, the name of which is used as the constraint reference in the corresponding test event. When an event (PDU or ASP) is received, it is decoded to a value E-node which only contains specific values. The expected constraint for this event is then loaded from disk for comparison by traversing the two trees. Since there is a constraint type in each E-node, the types other than specific values can be easily compared. For example, if

ASN.1 ASP Type Definition	
ASP Name:	T_CONNECTrequest
Comment:	Just for demo purpose
Type Definition	
SEQUENCE { toAddr OCTETSTRING, -- destination address fromAddr OCTETSTRING, -- source address qtsReq INTEGER -- quality of service requested }	

Table 1: An example ASN.1 type definition

ASP Constraint Declaration		
Constraint Name:	T_CONNECT_1	
ASP Type:	T_CONNECTrequest	
Comment:	Just for demo purpose	
Parameter Name	Parameter Value	Comments
toAddr	'89520A32'H	
fromAddr	'89520A28'H	
qtsReq	'52'H	

Table 2: Constraint declaration for the sample type

the flag indicates the field is an integer within a range, the value in the received field is checked against the range. In this way, all matching mechanisms defined in TTCN can be accommodated.

Figure 3: Value E-node tree for the example

6 Experiences

We have implemented a prototype of the ATG using the programming language C on a Sun workstation running the SunOS 4.1 operating system. It should be relatively easy to port the system to other platforms.

The system consists of three components as highlighted in Figure 1. The test sequence generator uses the combined testing approach described in Section 2 to produce a set of I/O paths. Each I/O path corresponds to a test case. The paths cover both the control and data flows of the protocol. During the process of I/O path creation, the constraint generator constructs the constraints associated with a test case using symbolic execution techniques as described in Section 3. The constraints are then passed to the constraint solver, which solves one constraint system (for one test case) at a time. If a constraint system cannot be solved, an error message is printed and the test case discarded. For unconstrained parameters, random values from the associated domains will be used.

The ATG is an important subsystem of the integrated protocol testing system developed at UBC [6]. We have applied the system to a number of real-life protocols, including ISO TP0, FDDI, and LAPB. The test cases generated are translated into an executable tree form for execution in our test executor. The test data generated from the constraint solver are translated into a binary form and saved on disk. The data are pointed at by constraint references in the test events. During test execution, the tester loads in a test case and executes it step by step. When an event contains a constraint reference, the corresponding test data are read for use in sending the event or comparing a receive event.

As an example, part of the test cases for TP0 generated by our test sequence generator are listed in the appendix (see [13] for a detailed Estelle.Y specification of TP0). Du-paths with the same I/O sequence but different variable values are merged into one du-path with a separate record for each variable. This record is used for testing different dataflow effects, i.e., for different test purposes.

Although the length of the final test suite cannot be guaranteed to lie within the upper bound for control flow testing given in Section 2 because there could be a large amount of data flow paths, the lengths were still relatively short in our experiments. The TPO results are listed in the appendix. The total number of test cases for TP0 was 14, consisting of 75 test steps. Of the constraints generated, only the output constraints for the first two test cases are listed due to space limitation. Note that input constraint corresponds to the input parameters while the output constraint restricts the output parameters for each test step. Test data that satisfy the constraints are then generated using the constraint solver. For example, for the event TCREQ:

```
TCREQ ::= SEQUENCE {
    toAddr    OCTETSTRING,
    fromAddr  OCTETSTRING,
    qtsReq    INTEGER
}
```

The values for `toAddr` and `fromAddr` are determined according to a specific test configuration. For example, suppose the initial domain of the parameter `qtsReq` is (-100:100). For test case 0, since `qtsReq` is constrained by `qtsReq >= 0`, the domain is reduced to (0:100) after domain reduction. A random value within the reduced domain, say 66, is then selected for `qtsReq`.

Constraints generated are saved on disk as a value E-node tree for use during test execution.

7 Conclusions

In this paper, we have presented a test suite generation tool set based on formal specifications. The tool set consists of three major components: test sequence generator, constraint generator, and

constraint solver. The tool generates test suite with appropriate test data for protocols specified as extended finite state machines. Although we have chosen to use Estelle as the specification language, the basic principles can be applied to other FDTs such as SDL and Lotos with interaction parameters. The tool uses a combined control and data flow testing scheme and is integrated into a comprehensive protocol testing environment. Automatic test data generation using a constraint solver is, to our knowledge, the first such work in protocol testing. We have applied the tools to a few real life protocols and found the tools both useful and practical.

References

- [1] S.T. Chanson, S.T. Vuong, and H. Dany. On multi-party and interoperability testing using the ferry clip approach. *Computer Communications*, April 1992.
- [2] S.T. Chanson and J. Zhu. A unified approach to protocol test sequence generation. In *Proc. IEEE INFOCOM*, San Francisco, March 1993.
- [3] L.A. Clark and D.J. Richardson. Application of symbolic evaluation methods. *The Journal of Systems and Software*, 5, 1985.
- [4] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
- [5] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9), September 1991.
- [6] S.T. Chanson et al. The UBC Protocol Test Environment. In *Proc. IWPTS'93*, 1993.
- [7] T. Higashino et al. Test system for a restricted class of LOTOS expressions with data parameters. In *IFIP Protocol Test System V (C-11)*, 1993.
- [8] Michael R. Garey and David S. Johnson. *Computers and intractability: A guide to the theory of NP completeness*. W. H. Freeman, 1979.
- [9] ISO/IEC 9646. *Information Technology – Open Systems Interconnection – Conformance testing methodology and framework, Part 3: The Tree and Tabular Combined Notation (TTCN)*. March 1991.
- [10] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, April 1985.
- [11] M. Sample and G. Neufeld. Support for ASN.1 within a protocol testing environment. In *Proc. IFIP 10th Symp. on Protocol Specification, Testing, and Verification*, 1990.
- [12] H. Ural. Test sequence selection based on static data flow analysis. *Computer Communication*, 10(5), 1987.
- [13] L. Zhou. A New Approach to Generating and Selecting Test Sequences for Conformance Testing. Master's thesis, Dept. of Computer Science, Univ. of British Columbia, November 1992.

Appendix: Test cases and constraints for TP0

(1) Test cases for TP0

Test Case 0: idle, wfcc, idle, wftr, idle, wfcc, idle

step 0: (TCREQ,-)/(-,CR)
step 1: (-,DR)/(NDREQ,-), (-,-)/(TDIND,-)
step 2: (-,CR)/(TCIND,-)
step 3: (TCRES,-)/(TDIND,DR)
step 4: (TCREQ,-)/(-,CR)
step 5: (-,DR)/(NDREQ,-), (-,-)/(TDIND,-)

Test Case 1: idle, wftr, idle, wfcc, idle

step 0: (-,CR)/(TCIND,-)
step 1: (TDREQ,-)/(-,DR)
step 2: (TCREQ,-)/(-,CR)
step 3: (-,DR)/(NDREQ,-), (-,-)/(TDIND,-)

Test Case 2: idle, wftr, idle, wfcc, idle

step 0: (-,CR)/(TCIND,-)
step 1: (TCRES,-)/(TDIND,DR)
step 2: (TCREQ,-)/(-,CR)
step 3: (-,DR)/(NDREQ,-), (-,-)/(TDIND,-)

Test Case 3: idle, wftr, data, idle, wfcc, idle

step 0: (-,CR)/(TCIND,-)
step 1: (TCRES,-)/(-,CC)
step 2: (TDREQ,-)/(NDREQ,-)
step 3: (TCREQ,-)/(-,CR)
step 4: (-,DR)/(NDREQ,-), (-,-)/(TDIND,-)

...

(2) Input constraints

Test Case 0

(step0: TCREQ.qtsReq) >= (0)
(step1: DR.disReason) == (0)
(step3: TCRES.qtsReq) <= (26)
(step4: TCREQ.qtsReq) >= (0)
(step5: DR.disReason) == (0)

Test Case 1

(step2: TCREQ.qtsReq) >= (0)

(step3: DR.disReason) == (0)

Test Case 2

(step1: TCRES.qtsReq) <= (11)

(step2: TCREQ.qtsReq) >= (0)

(step3: DR.disReason) == (0)

Test Case 3

(step1: TCRES.qtsReq) <= (11)

(step3: TCREQ.qtsReq) >= (0)

(step4: DR.disReason) == (0)

...

(3) Output constraints

Test Case 0

step0: CR.sourceRef = 300
step0: CR.option = 100
step0: CR.callingAddr = step0: TCREQ.fromAddr
step0: CR.calledAddr = step0: TCREQ.toAddr
step0: CR.maxTpdu = 80
step1: NDREQ.discReason = step1: DR.disReason
step1: TDIND.tsDiscReason = step1: DR.disReason
step1: TDIND.tsUserReason = step1: DR.addClearReason
step2: TCIND.toAddr = step2: CR.calledAddr
step2: TCIND.fromAddr = step2: CR.callingAddr
step2: TCIND.qtsPro = 26
step3: DR.destRefer = step2: CR.sourceRef
step3: DR.disReason = 0
step3: DR.addClearReason = 0
step3: TDIND.tsDiscReason = 0
step4: CR.sourceRef = 300
step4: CR.option = 100
step4: CR.callingAddr = step4: TCREQ.fromAddr
step4: CR.calledAddr = step4: TCREQ.toAddr
step4: CR.maxTpdu = 80
step5: NDREQ.discReason = step5: DR.disReason
step5: TDIND.tsDiscReason = step5: DR.disReason
step5: TDIND.tsUserReason = step5: DR.addClearReason

Test Case 1

step0: TCIND.toAddr = step0: CR.calledAddr

```

step0: TCIND.fromAddr = step0: CR.callingAddr
step0: TCIND.qtsPro = 11
step1: DR.disReason = 0
step1: DR.addClearReason = step1: TDREQ.tsUserReason
step1: DR.destRefer = step0: CR.sourceRef
step2: CR.sourceRef = 300
step2: CR.option = 100
step2: CR.callingAddr = step2: TCREQ.fromAddr
step2: CR.calledAddr = step2: TCREQ.toAddr
step2: CR.maxTpdu = 80
step3: NDREQ.discReason = step3: DR.disReason
step3: TDIND.tsDiscReason = step3: DR.disReason
step3: TDIND.tsUserReason = step3: DR.addClearReason

```

...