National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r.86/06)

Canada

An Interactive Test Sequence Generator

by
Russ Short

A M. Sc. Thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree M. Sc. in Computer Science.

University of Ottawa
Ottawa, Ontario, Canada
May 16, 1986.

© Russell Short, Ottawa, Canada, 1986.

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-36545-5

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

An Interactive Test Sequence Generator has been designed and implemented. ITSG provides an interface between a tester and a logic specification for the purpose of generating testcases from a given logic specification. A logic specification is a collection of Horn clauses coded in Prolog describing the externally observable behavior of a system. ITSG facilitates a controlled traversal of execution paths of the system described by the given logic specification and is written in C_Prolog running under UNIX 4.2 on VAX 750.

Logical implications of a specification are often implicit and difficult to understand and remember. ITSG keeps track of the logical implications and details of the protocol by following the given logic specification and making readily available to the tester specification facts and consequences of choices made earlier by the tester.

An example conformance test suite for Transport protocol Class 0 implementations has been designed and presented. Sample, representative testcases of this CTS which have been generated by ITSG are included.

ACKNOWLEDGEMENT

The author wishes to thank his supervisor Dr. H. Ural for his guidance and instruction in the field of protocols and for his efforts to impart to me the 'savoir faire' of technical writing. Many thanks also to my other teachers, particularly to Dr. Probert and Dr. Cheung and also to the professors and students who participated in the meetings of the PRG (protocol research group).

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In computer communication networks [Tane 81], the interactions between communicating entities are based on the exchange of messages. The orderly exchange of messages between such entities is governed by a set of rules which is called a communication protocol.

The International Organization for Standardization (ISO) defines a hierarchy of protocols in the Open Systems Interconnection (OSI) Reference Model [Zimm 80]. In this model, protocol layer N provides a particular communication service (i.e. N-service) to layer N+1. The N-service is specified in the N-service specification in terms of possible types and execution sequences of interactions (i.e., N-service primitives) exchanged between layer N and layer N+1. According to the OSI Reference Model, the N-service is provided by the collaboration of protocol entities (i.e., N-entities) in layer N over the (N-1)-service. That is, an N-entity participates in providing the N-service by communicating with other N entities via the (N-1)-service. The N-protocol specification describes the participation of an N-entity in the provision of the N-service in terms of its responses to requests (N-service primitives) from its users ((N+1)-entities), to messages (or Protocol Data Units (PDU's) from other N-entities received via the (N-1)-service, to control messages ((N-1)-service primitives) received from the (N-1)-service, and to internal events such

as timeouts. That is, the N-protocol specification is an abstract representation of the behavior of each implemented protocol entity in layer N. Thus, a complete implementation of an N-entity (i.e., an N-protocol implementation) may be constructed by successively refining the N-protocol specification.

Each protocol implementation should conform to the particular protocol specification that it implements in order to establish proper interworking between implementations of the same protocol specification. Implementations may be allowed to implement only a common subset of the overall functionality of the protocol. The subset implemented by an implementation is declared by its PICS (protocol implementation conformance statement). Implementations must then conform to the subset of the protocol specification which is claimed to be implemented in their PICS.

One way of accessing the conformance of a protocol implementation to a particular protocol specification is via protocol implementation testing. In the following subsection, the issues related to protocol implementation testing are briefly introduced.

1.1. Protocol Implementation Testing

The aim of protocol implementation testing is to demonstrate that a given protocol implementation conforms to the particular protocol specification. According to typical

proposed test architectures [Rayn 82], a protocol implementation is tested as a "black-box" over a communication medium between an assessment site and a client site, as in Figure 1. The assessment site configuration includes an "Active Tester" (AT). The client site configuration includes the protocol implementation under test (IUT) and a "Test Responder" (TR) acting as the user of the service provided by the IUT. During testing, the conformance of the IUT to the particular protocol specification is assessed through the observations of controlled sequences of interactions between AT and TR which are generated as a result of the application of a suite of testcases.

A testcase is a complete, independent specification of the actions required to achieve a specific test purpose. It is a sequence of input and output interactions that the IUT exchanges with its user and its peer, i.e., a sequence of input stimuli and corresponding expected responses.

One of the major subtasks in protocol implementation testing is the generation of effective testsequences. In software engineering, complementary testing techniques are used involving both black-box and white-box (structural) test generation strategies in order to improve the effectiveness of tests. However, a common assumption in protocol testing is that the internal structure (i.e., source code) of the IUT is not available [Rayn 82]. This assumption restricts the test sequence generation strategies to blackbox

or specification-based testing techniques; i.e, the design and internal structure of the IUT is not taken into account to guide the construction of the test sequences (and thus the testcases).

1.2. Motivation

In general, exhaustive testing (i.e., testing for all possible interaction (service primitive or PDU) sequences) is practically infeasible due to the values and variations of interaction parameter fields [Piat 80]. A practical compromise which attempts to reduce the size of the set of interaction sequences is that of selecting interaction sequences randomly (e.g. [LiMc 83]). However, random (test sequence) selection may justifiably be criticized from the viewpoints of completeness and effectiveness for software testing purposes [Myer79]. Thus, practical testing is confined to detecting particular instances of errors by using a manageably sized representative and discriminating set of interaction sequences. In such a set, each interaction sequence is considered to correspond to a number of functionally equivalent sequences in the set of all possible interaction sequences and is aimed to check whether the corresponding function is correctly implemented or not [MiHo81].

Disregarding values and variations of parameter fields of interactions, relatively simple Finite State Machine (FSM) based representations of given protocol specifications

are generated. Based on the state transition information of such representations, several FSM test techniques can be employed [SaBo 84]. However, the resulting test sequences mainly test for the control structure of the IUT. On the other hand, a test design methodology which is able to cope with interaction parameters requires considerable amount of effort in the case of complex protocol specifications [SBC 85]. Thus, it is desirable to provide a tool which facilitates interactive generation of test sequences from a given protocol specification.

1.3. Objectives and Contributions of the Thesis

Our primary objectives are to design and implement an interactive testsequence generator which provides an interface between a tester and a logic specification. This tool is used to facilitate a controlled traversal of the execution paths through the state space of the system being described by the given logic specification. ITSG's first two functionalities provide a a Help facility (i.e., static and dynamic protocol information accessing functions as well as on-line documentation of ITSG itself) and a protocol Processing facility (i.e., path generation and testcase construction capability). C-Prolog offers only rudimentary access to external files and so a third facility i.e. the management facility, is developed. These 3 facilities, i.e. help, processing and management, work harmoniously together providing the tester with information upon which to base

action, the capability to act and manage the output appropriate with testcase generation, respectively.

Our secondary objective is to demonstrate the feasibility of interactive generation of testcases using this tool such that the tester may make fast, correct, well-documented and complete testcases. We demonstrate the ease of use, the friendliness, and the flexibility of the generator on the example of constructing a conformance test suite for OSI Transport protocol class 0 implementations.

In chapter 2, we discuss the form of logic specifications which is the form ITSG expects the protocol to be in. Chapter 3 is devoted to ITSG itself; firstly an outline of its functionality, then an exhibition of its composition through a listing of the commands with explanations are given. This listing corresponds exactly with what a user gets on-line from ITSG's self-documentation aspect of the 'help' facility except the command syntax. Chapter 4 discusses the construction of conformance test suites (CTSs) with ITSG. The chapter begins with an introduction to CTSs for protocols, next discusses a strategy to make CTSs, then shows how ITSG functions allow the fulfillment of that strategy, then generates an example CTS from the Class 0 Transport protocol and lastly discusses the generation of testcases. Representative actual testcases are shown in Appendix C. In chapter 5, we draw our conclusions.

2. LOGIC SPECIFICATIONS

A logic specification [Sidh 83, UrPr 83] is a collection of Horn clauses coded in PROLOG [ClMe 81] describing the externally observable behavior of a system (e.g., a protocol entity, a service provider, etc.). A major advantage of logic specifications over other executable specifications is that logic specifications may allow for invertible execution. That is, they can be executed for both generating and recognizing the externally observable behavior of the specified systems [UrPr 84]. Thus, for example, a logic specification of a protocol entity can be employed as part of

i) a trace checker which determines whether the traces (i.e., execution histories) of a protocol implementation are acceptable interaction sequences and

ii) a test sequence generator which constructs interaction sequences for testing protocol implementations [UrPr 86].

Logic specifications are based on the interaction model given in [Boch 80] which characterizes distributed systems. In this interaction model, a system (or a subsystem) is considered as a collection of processes (e.g., modules) where each process is interacting with its environment (e.g., other processes) through its interaction points. An interaction between two processes takes place at a pair of corresponding interaction points (one in each interacting process) which is called a channel. An occurrence of an interaction between two processes involves a transfer of

information from one process to the other over a particular channel between these processes. Each interaction transfers information only in one direction. The process initiating an interaction considers that interaction as an "output" interaction which in turn is considered as an "input" interaction by the process receiving the transferred information.

Each channel between a pair of interacting processes may sometimes be associated with a pair of interaction queues, one for each direction of information transfer. Thus, the output interaction of a process at an interaction point is queued before it is considered as an input interaction at the related interaction point of the corresponding process. When the lengths of interaction queues are assumed to be zero, this scheme lends itself to modelling of rendezvous type of interactions between interacting processes. In this case, an interaction between a pair of processes occur when both are ready to execute the interaction (one as an input and the other as an output interaction) at their related interaction points.

A logic specification of a process describes the externally observable behavior of the process (i.e., the possible orderings of interactions exchanged at the interaction points of the process) in terms of its state space and possible state transitions. The state space of the process is specified by a set of variables. A possible state of the

process is characterized by the values of these variables. Each state transition is defined by an enabling condition which is a combination of boolean expressions involving state variables and possibly an input interaction and by an action which may update some state variables and may generate some output interactions. Accordingly, the logic specification of a process is a set of clauses which consists of two parts; namely, transitions part and extensions part.

2.1. Transitions Part

Each clause in this part defines a state transition and is in the form of:

```
t_name(TID,Present_state,Next_state,Input,Output) :-
enabling_condition , actions.
```

where name stands for the name of the process and TID identifies the transition. Here, Present_state and Next_state are two lists containing values of state variables before and after the execution of the transition; Input and Output are lists representing interactions of the process taking place during the transition; enabling_condition is the set of predicates which must hold for the transition to take place; and actions which define some relationships between mainly present and next values of state variables.

Each state variable is an element of both Present state and Next state lists and has the same relative position in each list. Moreover, each state variable is represented by

the same variable name in both lists except those variables whose values are updated during the execution of the transition. For these variables, the corresponding elements in both lists should contain different names. For simplicity, we use the following convention: if the value of a state variable is updated then the name of the variable is preceded by a "P" in the Present_state list whereas it is preceded by an "N" in the Next_state list.

For each interaction point of the process, there is one element in input and output lists. Each element of the Input (Output) list is an input (output) interaction and the relative position of an interaction in each list identifies the interaction point over which the interaction occurs. Each interaction is represented by
a list containing
the interaction point identifier,
the interaction identifier, and
the values of interaction parameters.

The contents of Input and Output lists for each transition are determined as follows :

- All elements of the Input list are empty lists (denoted by [] in Prolog) except the element corresponding to the interaction point over which the interaction referred to in the transition occurs. In the case that the transition is a spontaneous transition (i.e., the one with no input interaction), all elements of the

Input list are empty lists. The non-empty element of the Input list (if any) is a list itself, containing the interaction point identifier, interaction identifier, and a variable representing the input interaction parameter list (i.e., IPL).

- All elements of the Output list are empty lists except the element(s) corresponding to the interaction point(s) over which the output interaction(s) referred to (if any) in the transition occur. The non-empty elements of the Output list are lists themselves, each containing the interaction point identifier, interaction identifier, and a variable representing the output interaction parameter list (e.g., OPL).

The components of the enabling condition are the following:

- A predicate in the form of IPL=[P1,P2,...,Pn] which is used to decompose the variable IPL representing the parameter list of the non-empty element (if any) of the Input list into its components P1,P2,...,Pn. In order to determine the values of the parameter fields, list IPL is passed as a parameter to a relation called input which is defined in the extensions part of the logic specification. Note that if none of the input interaction parameters is referred to in the transition clause, it is not necessary to spell out variable names corresponding to the elements of list IPL, and thus

this predicate can be eliminated. P- The remaining component of the enabling condition is a boolean expression which is constructed by using built_in Prolog relational and logical operators. Note that expressions on either side of a relational operator in Prolog should be a single term (i.e., a variable or a constant). Unlike the other relational operators, "=" can be used to assign a value to a variable. If the variable on the LHS or the RHS of "=" is uninstantiated then it takes the value (of the variable) on the opposite side (e.g., the above predicate which is used to decompose list IPL).

The actions performed during the operation of the transition are :

- Assignment of values to state variables and to parameter fields of an output interaction. In principle, the built_in Prolog predicate "is" can be used to perform any assignment. However, in order to maintain the invertibility of the resulting logic specification, assigning a value to a parameter field of an output interaction is performed using the built_in Prolog predicate "=" in the following manner: <variable denoting the parameter field> "=" <variable_v> where <variable_v> corresponds to the value of the Prolog equivalent of an expression_e which is obtained in a step_wise manner using built_in Prolog predicates

before it is used in this predicate. On the other hand, assigning a value to a state variable (i.e., a variable in the Next_state list) can be performed by using the predicate "is". This is based on the assumption that logic specifications need not be executed by supplying the final state and asking for an initial state.

- Generation of output interactions which includes decomposition of the variable OPL representing the parameter list of a non-empty element (if any) of the Output list into its components R1,R2,...,Rn by the following predicate: OPL=[R1,R2,...,Rn]. In order to determine the values of the parameter fields, list OPL is passed as a parameter to a relation called output which is defined in the extensions part of the logic specification. Note that if none of the output interaction parameters is referred to in the transition clause, then it is not necessary to spell out the variable names corresponding to the elements of the list OPL, and thus this predicate can be omitted from the clause.

2.2. Extensions Part

The extensions part of a logic specification contains definitions of relations referenced in the transitions part. These relations are "input", "output", and all other relations referred to in the definitions of "input" and "output". Relations "input" and "output" are intended to

generate and recognize valid input and output interactions, respectively. They are both of the same form, i.e., they are defined as collections of clauses corresponding to the interactions that the process exchanges at its interaction points. There should be one clause in the relation "output" for each interaction identified as an interaction which may be initiated by the process and one clause in the relation "input" for each interaction identified as an interaction which may be initiated by the environment of the process. Each clause defined as part of the relation "input" (or similarly of the relation "output") is of the form :

```
input([interaction_point_id,interaction_id,PARAM)
     :- interaction_id(PARAM).
```

where PARAM stands for the list of interaction parameters and interaction_id is the name of the relation which defines the parameter fields (i.e., format) of the corresponding interaction in the following form:

```
interaction_id([P1,P2,.....,Pn])
     :- name_of_the_1st_parameter_field(P1),
        name_of_the_2nd_parameter_field(P2),
        ...,
        name_of_the_nth_parameter_field(Pn).
```

where each condition referred to in the above clause is a reference to a clause defining the values of the corresponding interaction parameter. In order to maintain the invertibility of the resulting logic specification these clauses are defined in the following form :

```
name_of_parameter(X)
    :-(var(X),assign_value_of(X))
     ;(nonvar(X),check_value_of(X)).
```

where var (nonvar) is a built-in predicate of Prolog which

ensures that variable X is uninstantiated (instantiated). Thus, if X is uninstantiated, a valid value is assigned to X otherwise its value is checked to see whether it is a valid value.

2.3. A Logic Specification Example

Logic specifications of protocol entities and service providers can be derived from the corresponding formal or informal protocol and service specifications, respectively. A procedure for constructing these logic specifications from the original protocol and service specifications given in Estelle (i.e., Extended State Transition based formalism of ISO [ISO 85a]) is proposed in [Ural 86]. This procedure assumes that each original specification is given in the form of a single-module specification in Estelle describing the externally observable behavior of a module (e.g., process) in terms of "normal form transitions". A normal form transition consists of an external input interaction (optional), explicit present and next state identifiers, an enabling condition, and an action defining a single execution path and possibly an external output interaction [SaBo 85]. Although specifications of protocol entities and service providers in Estelle may be given in terms of externally observable behaviors of possibly more than one module, the application of various transformations given in [SaBo 85] transforms these specifications into single-module specifications containing normal form transitions.

For example, a Transport protocol specification [ISO 82] given in Estelle describes the behavior of a transport protocol entity (denoted tpe) which consists of a mapping module and an arbitrary number of abstract protocol modules (ATPs). By applying the transformations proposed in [SaBo 85], a single-module specification of the Transport protocol entity can be obtained from this specification. A logic specification of the protocol entity may then be constructed from the resulting single-module specification by following the procedure given in [Ural 86]. Accordingly, each clause in the transitions part of the resulting logic specification is of the form :

```
t_tpe(TID, PS_tpe, NS_tpe, [I_tsap, I_nsap],[O_tsap, O_nsap])
     :- enabling_condition, action.
```

Here, "t_tpe" stands for the name of the relation defining transitions of the transport protocol entity (tpe), and TID stands for the transition identifier. PS_tpe and NS_tpe are two lists representing the present and next values of state variables. Both of these lists consist of the following elements:

major state variable	
transport connection end point identifier	(TCEP)
TPDU size	(TPDU_S)
quality of transport service	(QOS)
buffer(for data fragments received) from user	(BFU)
buffer(for data fragments to be sent) to user	(BTU)
sequence number of the last data item sent	(SSEQ)
sequence number of the last data item received	(RSEQ)
local reference	(LREF)
remote reference	(RREF)
calling T_address	(CGTA)
called T_address	(CDTA)

The input (or output) list contains two elements correspond-

ing to the input (or output) interactions exchanged at transport service access point (denoted tsap) and network service access point (denoted nsap), respectively.

The following is an outline of this logic specification. The complete logic specification of the Transport protocol entity is given in Appendix A.

```
t_tpe(1,[idle,       PTCEP, PTPDU_S, PQOS, BFU, BTU, SSEQ, RSEQ,
     PLREF, RREF, PCGTA, PCDTA],
       [wait_for_cc,NTCEP, NTPDU_S, NQOS, BFU, BTU, SSEQ, RSEQ,
     NLREF, RREF, NCGTA, NCDTA],
       [[tsap, tcreq, IPL], []],
       [[], [nsap, cr, OPL]]) :-
     /*  acceptable_QOS  */
     /*  tsap is the transport service access point  */
     /*  tcreq is the input interaction identifier   */
         IPL = [ NTCEP      /*  Value of TCEP  */
                ,NCDTA      /*  Value of to_T_address  */
                ,NCGTA      /*  Value of from_T_address  */
                ,NQOS       /*  Value of Quality_of_service  */
                ,'normal'   /*  Options_proposed  */
                ,TSDATA     /*  User data  */],
     input([tsap, tcreq, IPL]),

     /*  next values of various state variables have been
         determined as the values of some of the input
         interaction parameters */
     /*  nsap is the network service access point  */
     /*  cr is the output interaction identifier   */

       OPL = [CREDIT      /*  Credit vale  */
             ,NLREF       /*  local reference  */
             ,TSDATA      /*  User data  */
             ,0           /*  Class indication  */
             ,'normal'    /*  Options_indication  */
             ,NCGTA       /*  Calling_T_address  */
             ,NCDTA       /*  Called_T_address  */
             ,NTPDU_s     /*  TPDU_size_indication  */],
     output([nsap, cr, OPL]),
```

```
        /*  At this point next values of other relevant state
            variables have been determined as the values of
            corresponding output interaction parameters  */

        NTPDU_S > 0.

        /*  The value of the TPDU_size_indication parameter of
            the output interaction should be at least '128' */

    t_tpe(2,...) :- .......
     :
     :
    t_tpe(n,...) :- .......

    input([tsap, tcreq, PARAM]) :- t_connect_req(PARAM).
    input(...) :- ...
     :
     :
    input(...) :- . . .

    output([nsap, cr, PARAM) :- cr_tpdu(PARAM)..
    output(...) :- ...
     :
     :
    output(...) :- ...

    t_connect_req([P1, P2, P3, P4, P5, P6]) :-
        tcep(P1)
       ,to_T_address(P2)
       ,from_T_address(P3)
       ,quality_of_TS(P4)
       ,options(P5)
       ,ts_user_data(P6).
     :
     :
    cr_tpdu([P1, P2, P3, P4, P5, P6, P7, P8]) :-
          credit(P1)
         ,source_reference(P2)
         ,user_data(P3)
         ,class_indication(P4)
         ,options_indication(P5)
         ,calling_address(P6)
         ,called_address(P7)
         ,tpdu_size_indication(P8).

     :
     :
    /* Definitions of parameter field (i.e., their values) */
```

The transition defined by the first clause above is between states "idle" and "wait_for_cc" and involves the

reception of a "tcreq" at tsap and the transmission of a "cr_tpdu" at nsap.

2.4. Logic Specifications of Composite Systems

A system may be specified as a composition of several subsystems where each subsystem interacting with its environment (i.e., other subsystems) through its interaction points. For simplicity, consider each subsystem as a single process. Then, by using the given logic specifications of its component subsystems (i.e. processes), a logic specification of a composite system is built by [Ural 86]:

- defining the system state as a composition of the states of all the subsystems.
- determining the interaction points at which the external behavior of the composite system is observed. The interaction points of the system are those interaction points of the subsystems that interact with the system environment.
- specifying system transitions in terms of transitions of the subsystems. For simplicity, we assume that one and only one subsystem transition takes place during a system transition. That is, each system transition corresponds to a transition in one of the subsystems. This assumption results in a set of clauses where each clause defines a system transition in terms of a single transition of one of the subsystems. , Thus, the number of clauses defining all possible system transitions becomes the same as the number of subsystems in the system.

- constructing the transitions part of the resulting logic specification as the combination of transitions parts of the logic specifications of all the subsystems and the set of clauses defining the system transitions.
- forming the extensions part of the resulting logic specification as the union of the extensions parts of the logic specifications of all the subsystems.

The definition of the system state and the construction of clauses defining system transitions of a composite system differ in some aspects depending on whether the subsystem interconnections are modelled in terms of interaction queues or rendezvous: If the subsystem interconnections are modelled in terms of interaction queues then the system state is composed of the states of all the subsystems and the contents of all the interaction queues. On the other hand, when the rendezvous type of interactions are used to model the subsystem interconnections, an outstanding interaction replaces each pair of interaction queues in the definition of the system state. We consider an interaction outstanding if it is initiated but not received and thus not completed yet. Details of the construction of logic specifications of composite systems can be found in [Ural 85].

3. THE INTERACTIVE TEST SEQUENCE GENERATOR

ITSG provides an interface between a tester and a logic specification mainly for the purpose of generating test-sequences. It is written in C_Prolog under UNIX 4.2 Operating System on a VAX750. As shown Figure 2, it also provides the capability of accessing to C-Prolog and Unix as well as to several external files such as 'erroneous values files' where erroneous values for interaction parameters are stored, files of paths generated by traversing the state space of the Logic Specification (which are the source for testcase generation), 'testcase files' where testcases generated by the tester via ITSG and supplemented with control information and comments are saved, etc.

The basic functionality of ITSG is to facilitate a controlled traversal of the execution paths of the system described by the logic specification. Each execution path between a pair of system states may be expressed (represented) as a sequence of

(a) transition identifiers or

(b) Input/Output interaction identifiers or

(c) Input/Output interactions (i.e. including interaction parameter values).

In fact, each execution path corresponds to a possible ordering of externally observable interactions of the system

which is stated explicitly in the cases b and c and implicitly in case a. Thus, we identify three basic modes for ITSG, namely: TID (transition identifier) mode, IID (interaction identifier) mode and I (interaction) mode.

The controlled traversal of execution paths is provided by ITSG via various functions which can be classified as help/display facility, protocol processing facility, and management facility. These functions are introduced in the following sections briefly.

3.1. The Outline of ITSG's Functionality

ITSG allows a tester to commence generating a set of testcases with an abstract formulation of this set to be generated and to defer details of implementing each testcase in this set to ITSG. Current manual techniques necessitate a tester to have the basic 'modus operandi' in his mind and then to set about implementing this strategy manually while being careful to follow the specification and to also avoid making errors in spite of the profusion of data. Logical implications of the specification are often implicit and difficult to understand and remember. ITSG keeps track of the logical implications and details of the protocol by following the given logic specification and making readily available to the tester specification facts and consequences of choices made earlier by the tester.

In the following subsections we briefly introduce the

functionality of the ITSG in terms of the functions provided to the tester.

3.1.1. Help/Display Facility

At the most elementary level of the help facility, ITSG provides an on-line listing of ITSG commands and their formats. A novice user can access this information simply by typing 'help'. An experienced user can get help directly on any of the 3 facilities of ITSG. Secondly, there is 'help(ID)' which allows viewing of logic specification information. For example 'help(TID)' will display the whole transition clause with that TID (transition identifier). As well, the list functions can get protocol state information such as a list of specified receptions, a list of unspecified receptions, the present state, the present status, etc.

3.1.2. Protocol Processing Facility

With all the information of the logic specification instantaneously available in an organized fashion, the user next needs protocol processing functions (path generating functions and testcase construction functions, etc.). There are two user-visible path-generating functions: They are 'auto_fire' for automatic searching for paths and 'fire_one' for manual path generation. 'Auto_fire' employs a loopless traversal (with an optional path count halting mechanism) and 'fire_one' allows complete freedom in path generation with a backtrack function.

The testcase construction function is called 'mak_tc'. It is a composite of many other functions (e.g. "fire_all", "locate", etc.) and employs paths (generated by the path generating functions mentioned previously) as inputs and external files of completed testcases as outputs. This function has no parameters but receives its input and output directions from the user who is prompted for information. This function guides the tester in the generation of testcases with formatted output features such as testcase headers and 'pretty-printing' of testcases. The tester may append his/her own comments and verify his/her results if s/he wishes. It has relevant error diagnostics features and is very efficient.

Each testcase is intended to test either the correct functionality or defensive behavior of the IUT. In generating a testcase to test a particular correct, varied behavior, or a defensive behavior of the IUT, 'mak_tc' is instructed to interrupt (i.e. 'break') execution and allow any ITSG function to be employed - usually for information gathering and generating a correct, varied, or erroneous input (i.e. a PDU or a service primitive).

3.1.3. Management Facility

ITSG's management functions provide a degree of sophistication wherein the ITSG user can interact with the logic specification within the Prolog and Unix environment. Thus, ITSG allows accessing external files, processing internal

information, and also creating permanent external files for the end results (i.e., testcases). 'mak_tc' (discussed above) is a prime example of a function which operates unnoticeably in both Prolog and Unix environments. As well, the ITSG user can manually and comfortably manage external files via ITSG's management functionality as if they were part of a large internal Prolog database (i.e., a collection of Horn clauses).

Unix's 'ls' function has been transported to ITSG. This function reveals the directory of files in the external environment (see section 3.3 for details). Files or parts of files can be listed with 'listall'. They can be searched for clauses which can then be executed or manipulated. This is, for example, how numbered paths are accessed. Clauses can be loaded into the internal database with commands like 'load', 'loadall' and 'locate' and dumped out to files from the internal database with the command 'dump'.

3.2. ITSG Help Functions

The set of functions comprising ITSG's help facility provide the tester with various information s/he needs about the logic specification and about ramifications of the logic specification. These functions are briefly introduced below:

.help

Displays the three commands for accessing information about the three facilities of ITSG i.e. help/display facility, processing facility and management facility.

.help(initial_status)

Displays the initial status which includes the initial system state and the static status information often found in boolean conditions as boolean predicates such as able_to_provide_service, congestion, etc.

.help(present_status)

Displays the present status which includes the present system state and the static status information as in the initial status but which may be different because the tester might have changed some of the static status information.

.help(initial_state)

Displays the initial 'system state' which is a list containing initial values of the major and additional state variables.

.help(present_state)

Displays the present 'system state' which is a list containing current values of the major and additional state variables.

.help(mode)

Displays the present mode of ITSG (e.g., TID, IID, or I mode).

.help(ID)

Displays the Prolog clauses corresponding to a particular transition, interaction, or interaction parameter field (subfield) whose identifier is denoted by ID. Thus, the tester is provided with information about the particulars of a transition (eg., its enabling predicate or actions); about the format of an interaction (eg., the interaction parameter field names); about the subfields of a parameter field; or about the particular values of a field.

.lt

Lists the complete set of transition identifiers together with the system states at which they are defined in the logic specification.

.lf

Displays the list of identifiers of transitions that are firable at the present system state.

.lnf

Displays the list of identifiers of transitions that are not firable due to the current values of present

state variables.

.lsr

Lists the specified receptions for the present system state. Each specified reception is displayed as a list of 3 elements: [Transition identifier, Next system state, Name of the input interaction]. The tester may issue 'help(Interaction identifier)' and get the parameters of the interaction which is necessary information in generating erroneous or atypical values in a transition. Such values are generated by setting a 'break' on one of the parameters (see function 'mak_tc' in section 3.3). When the program halts at the 'break' the tester can insert an erroneous or atypical value (via the 'pe' or 'atyp' functions; refer to section 3.4) or get information and reset the 'break' on a subfield (via the 'add_break_clause' and 'mouse' functions; refer to section 3.4).

.lur

Displays the list of unspecified receptions at the present system state. These may be used to generate unspecified reception errors at path error points. Such errors are effected by issuing an 'ur' at the 'break'.

3.3. ITSG Processing Functions

These functions provide the capability of processing

through the state space as opposed to looking up information. Here input and/or output interactions are generated, test-sequences are composed and included into a Conformance Test Suite (CTS) to be used in future conformance testing of implementations of the protocol.

.initialize_state

Sets the system state to the initial state.

.initialize_status

Sets the present status to the initial status.

.set(present_state)

Allows the user to interactively set the present system state.

.set(present_status)

Allows the user to interactively set the present status.

.set(mode(M))

Sets the mode of ITSG to the mode denoted by variable M (i.e. 'tid', 'iid', or 'i').

.set(tc_template)

Allows the interactive setting of the testcase template. The testcase template designates the input

filename, functor, and number which causes the automatic retrieval of a prolog clause containing the path to be traversed and as well the name of the output testcase file and whether the testcase is to be in 'pretty print' format or 'new_line' format. From the example testcases of Appendix C two testcases (i.e., files) use the 'new_line' format. They are "example_TCs/tc_1_connection/mar_20_932_pm" and "example_TCs/tc_3_ur/mar_27_647_pm". All the other examples use the 'pretty-print' format.

.auto_fire(Final_state,No_of_paths,[Output_type,Identifier])

This automatically finds (searches) paths from the present system state to a Final state. The number of paths to be generated can also be specified. Auto-fire has a cycle detection and avoidance capability. It is mainly intended to be executed when ITSG mode is TID. The resulting execution paths may be displayed on the screen, stored internally (as part of the logic specification) or externally as a file. If it is desired to have the resulting paths stored, then the paths are numbered and assembled as assertions whose functors are the given Identifier.

.fire_one(ID)

This function allows firing a transition specified by ID which may be an identifier of a transition or input

interaction. The result is a change in the system state and augmentation of the execution path formed so far. Note that the execution path may be formed as a sequence of transition identifiers, input/output interaction identifiers, or input/output interactions depending on whether ITSG is in 'tid', 'iid' or 'i' mode, respectively.

.fout([Output_type,Identifier])

This function outputs the path (constructed by using a series of 'fire_one' transitions) with the given Identifier as functor. The only parameter of this function is interpreted exactly as the third parameter of 'auto_fire' explained above.

.backtrack(N)

Undoes N transitions and sets the present system state to the one where the last undone transition originates.

.fire_all(Execution_path,[Output_type,Identifier])

This function allows traversing the execution path given as a sequence of transition identifiers or input/output interaction identifiers. This function is intended to be employed when ITSG is in I mode. The interpretation of the second parameter is the same as the third parameter of 'auto_fire' explained above.

.mak_tc

This function is the main testcase construction function. Its name is short for 'make testcase'. It has no arguments because the tester is guided and prompted through the construction of testcases. It requests which path is to be used, calls 'fire_all', stops at the error in the path if there is one or prompts for the 'break' spot otherwise. The tester can decide to not put in a 'break'; of course that is the case for typical value generation. At the break points s/he calls 'ur', 'atyp' or 'pe' (or does a 'mouse' to extend the 'break' downstream) and with cntrl_d continues execution till the testcase is completed.

3.4. ITSG Management Functions

.set(trace_all) and reset(trace_all).

This function records the complete history of 'auto_fire' executions while on. Since auto-fire keeps only a successful path the tester does not know the paths attempted and later discarded. Trace_all will keep a copy of all transitions fired including those that were subsequently back-tracked.

.i_clean(In,Out)

This function is a filter which cleans an interaction sequence (testcase) of empty input and output symbols

inserted during interaction filling by the Logic Specification model to indicate the absence of input or output.

.add_break(N,Path,Result_Path)

This function inserts a 'break' in the path (a list of transitions) after the Nth transition. At the break point, the tester is consulted to insert the modification. For errors this function is called automatically but for generating atypical values for parameter fields the user may select where the 'break' in the path is to be inserted.

.add_break_clause(H,T,N,Result_Tail)

This function inserts a 'break' into the clause's tail which would be difficult were this function not provided. The clause head is H and the orginal clause tail was T. The 'break' is inserted after the Nth element of the clause tail 'T' resulting in 'Result_Tail'.

.dump(Filename,Functor)

This function retracts all relations with the given functor and writes them to the external file denoted by Filename.

.load(Filename,Head,Tail)

This function copies a clause from an external file

'Filename' into the internal database. Head and Tail represent the clause to be fetched from the file.

.loadall(Filename,Head,Tail)

Similar to the above function except all relations matching to the clause denoted by the Head and Tail pair are loaded into the internal database.

.locate(Filename,Head,Tail)

This function is similar to the 'load' function above except the clause from the external file is not asserted to the internal database but is expected to be used immediately.

.listall(Filename,Functor)

This function is used to view clauses with functor denoted by 'Functor' from an external file.

.listall(Filename)

This function lists all clauses in an external file.

.atyp(Head,Tail)

This function is to be used during a 'break' to add an atypical value.

.ur([Access_point,Reception_id])

This function creates an unspecified reception error.

.pe([Access_point,Reception_id],Head,Tail)

This function creates a parameter error.

4. THE CONSTRUCTION OF CONFORMANCE TEST SUITES

A conformance test suite (CTS) is a set of testcases to be employed for protocol implementation testing. Basically, a CTS may be composed of testcases that test the correct functionality of the IUT as well as its defensive behavior when subjected to erroneous and exceptional stimuli. The CTS for a particular protocol is based on the corresponding protocol specification. This specification states the correct functionality of the IUT. The testcases testing the defensive behaviour of the IUT can be constructed from the given protocol specification by systematically degenerating this partially complete specification. Equivalently, we may say that, this is accomplished by completing the specification and during the firing of the error transitions using the following error types. Three cases of degeneration may be specified as follows:

(1) unspecified receptions which are unspecified input interactions for the present system state.

(2) format errors which are generated by inserting an incorrect or unidentified field somewhere in the input interactions. Omission(s) of non-optional fields also constitute format errors.

(3) erroneous values which are generated by selecting out of range values, non-member values, etc.

Note that by completing the specification we still have strict constraints. For example, we would still wish only one error per testcase. That error is to occur precisely at the point in the path where the error transition is placed and what occurs consequently must conform to what the specification states will occur. Error generation for defensive testing may not at all be considered arbitrary.

ITSG provides an interface to a logic specification which may not be a complete specification (i.e., all error receptions at every state may not be specified). From such a specification a usually infinite set of paths, due to the presence of cycles in the specification, can be generated by starting from and returning to 'idle'. (Assume 'final' and 'error_state' are equivalent to a non-active 'idle' state). Each of these paths corresponds to a sequence of interactions called a testcase. Due to variations of interaction parameters and their values, the number of testcases may be very large. Thus, a CTS generated by ITSG should be a subset of this large set of all possible testcases. Naturally, the actual CTS selected should be some practical cover of the specification. (see Figure 3).

4.1. Criteria For A Standard Protocol CTS

According to [ISO 85b] a full conformance test suite, for a particular protocol, should be capable of testing all mandatory and optional features and functions over the maximum range of parameters and variations. This includes:

(1) testing for all mandatory and optional features of the protocol and their feasible combinations allowed by the static conformance requirements. Here, by the static conformance requirements we mean those requirements defining the kernal sets of capabilities of the implementation.

(2) testing for the major protocol phases and their various combinations such as connection establishment, data transfer, and connection termination for each feature of the protocol.

(3) testing for a range of variations in the following domains:

    (1) sequence variations

    (2) timing variations

    (3) PDU encoding variations

    (4) parameter variations in PDUs

The same document makes the following suggestions for domain variations:

- support of all PDU and service primitive types and all structural (taxonomic) variations of each type.

- 'normal' or default values for each parameter on each PDU and service primitive.

- boundary values plus at least one mid-range value for each integer parameter.

- for bitwise parameters, as many values as is practical, but not less than all of the 'normal' or common values,

- for interdependent pairs of PDU and service primitive parameters, 'critical' value pairs (representing multi-dimensional boundaries) and one 'normal' value pair.

(4) testing for the defensive behaviour of the IUT:

- all sequences of PDUs and service primitives where one PDU and service primitive is 'out of sequence' with respect to the defined protocol.

- at least one invalid PDU and service primitive type.

- at least one invalid value for each PDU and service primitive parameter, where such invalid values exist.

- all defined timers should be exercised, i.e. allowed to expire at least once.

[ISO 85b] continues discussing erroneous behavior and comments that further study is required. As we shall note later on, ITSG covers all the above cases and considerably widens the prospects for the generation of primitive and PDU format errors which were referred to above as invalid PDU and service primitive types.

4.2. Strategy To Fulfill The Standard CTS Criteria

ITSG allows the tester the freedom to generate any test with accuracy and with relative ease. We present an over-simplified and straightforward example of the generation of a Conformance Test Suite (CTS). This CTS avoids consideration of any particular protocol and could be automated since it is based on the simple concept of enumeration. It is not intended to replace the tester but only to highlight higher level considerations of ITSG. We partition the specification as implied by Figure 4 and generate tests that cover (enumerate) a collection of sub-partitions of each partition. For our generic example we construct our partition as follows:

(1) Specified receptions with no format errors and no value errors (i.e. no errors and default (typical) values only).

(2) Unspecified receptions.

(3) Format errors.

(4) Values which are atypical but valid and values which are invalid.

We will consider one case only for each of the partitions 1, 2 and 3. However, for partition 4, we consider the following cases:

valid_high,

valid_low,

valid_intermediate, and

invalid.

For each of the resulting cases we provide an explanation of how ITSG may be used to generate the particular subset of the CTS which corresponds to that case.

We mention that this simple strategy for a CTS explained here is based on 'enumeration' and so it is possible to do this automatically. We have left higher order, more semantic testing such as functional testing to the expert tester and engaged only in showing that tests can be as easily generated as traversing a FSM. Defensive testing is accomplished by means of error transition(s) and 'breaks' to halt and redirect the automatic generation of interactions.

1/ specified receptions + correct formats + default values

eg. a tour that covers the FSM i.e., all transitions fired once at least with all correct values. In this case (i.e. no errors at all), the tester may cover the basic interconnection testing, the functional range testing, and some of the dynamic range testing [Rayn 85]. To make such a subset of the test-suite the tester would traverse the FSM by the 'fire_one(ID)' function, using 'list_firables' and 'list_specified_receptions' to get information about the

possible next transitions, saving the path automatically and employing 'backtrack(N)' when needed. When finished 'fout' may be used to output the path to the screen, to save in the database or to save at an external file. For any complete path the tester has access to, he may call 'mak_tc' and be guided to generate a testcase. Many testcases may be generated from one path. ITSG will prevent the user from generating faulty testcases. As well ITSG's 'mak_tc' function generates a testcase with a header which provides the information about the path, a trace of relevant aspects of the process of generating the testcase thereby revealing the test purpose and lastly the value of the testcase printed in prolog. The 'help' and list functions could be employed as the tester wishes. 'Auto-fire' might also be used to complete a subpath generated by successive executions of 'fire_one' or to search for paths automatically. Our imaginary tour of our imaginary EFSM is very shallow but can serve to illustrate the manner that the tester generates correct and default-valued dynamic conformance testing. To ensure a tour, the tester may start with a list of all transitions and delete them as they are used and quit only when the list is empty.

2/ unspecified receptions + correct formats + default values

eg. a tour on which all receptions are used incorrectly at least once (by enumerating the input interaction identifiers

so every one has been invalid at least once). This tour is also not complete for dynamic conformance testing in general but it illustrates testing the defensive behaviour of the IUT with respect to unspecified receptions. The tester identifies unspecified receptions at the current system state by issuing a 'list_unspecified_receptions' (lur) and selects one of the unspecified receptions which is placed in the relation "input" in the logic specification to generate the unspecified reception.

3/ specified receptions + incorrect formats + default values

eg. a tour on which each parameter field or sub-field is replaced with an erroneous parameter field or sub-field. The receptions are all specified and the values are all correct but there exists an error in a format of a parameter field or sub-field. Information about the format of an input interaction can be requested at any time to facilitate parameter replacement.

4/ specified receptions + correct formats + upper boundary values

The tester sets each parameter field at least once to its upper boundary value while all other fields take default values.

5/ specified receptions + correct formats +

lower boundary values

Same as above except lower boundary values are used.

6/ specified receptions + correct formats + intermediate values

Same as above except intermediate values are used.

7/ specified receptions + correct formats + invalid values

Same as above except invalid values are used.

Having established a broad definition of a CTS in section 4.1 and shown how to fulfill that definition in section 4.2, we now introduce in the following section ITSG's anomaly making functions and illustrate a construction process of a simple CTS for a particular protocol, namely ISO class 0 transport protocol.

4.3. An Example CTS For The Transport Protocol Class 0

ITSG has 3 basic functions to generate anomalies during the construction of a CTS, namely 'ur' (unspecified receptions), 'atyp' (atypical parameter values), and 'pe' (parameter errors). Typical values with everything correct is the default case for ITSG and is referred to as typicals. Typicals and atypicals are not for defensive testing but unspecified receptions and parameter errors are. In section

4.2 we introduced 4 sub-partitions which were expanded to 7 cases.

The original 4 partitions were typicals, unspecified receptions, format errors, and, finally value variations (erroneous and correct). The 7 cases arose from the expansion of sub-partition number 4 (i.e., value variations) to 4 separate cases, i.e., 3 correct value cases (i.e., values high, values low and values intermediate) and 1 erroneous value case. These 7 cases are handled by ITSG's 4 functions. Their ordering is default for typicals, 'ur' for unspecified receptions, 'pe' for format errors, 'atyp' for high values, low values and intermediate values and 'pe' again (i.e., just as in formats) for invalid values.

We now provide a table as a convenience to understanding and demonstration of ITSG coverage of the essentials of CTS generation.

(top of section 4.2) Figure 4 sub_partitions	(body of section 4.2) 7 cases	(section 4.3) ITSG commands
typicals	typicals	default
unspecified receptions	unspecified receptions	ur
format errors	format errors	pe
values	high,low,intermediate	atyp
"	erroneous values	pe

4.3.1. Test Suite Plan

Initially our approach to generating the CTS consists of two phases; namely, connection establishment and data transfer phases. After having generated testcases

corresponding to typicals, we will abandon this distinction because the protocol is simple and there is no benefit treating these two phases separately for the generation of testcases corresponding to the remaining criteria.

Testcases begin at the 'idle' state and are made according to four criteria:

(typicals)

Generating testcases which end at the 'final' state using transitions with typical values. ITSG generates typicals by default.

(atypicals)

Generating testcases which end at the 'final' state using transitions with parameter variations i.e. atypical values. Note that non-erroneous timer variations would be included here. ITSG generates atypicals with the function 'atyp'.

(unspecified receptions)

Generating testcases to test defensive behavior using a valid error transition (called 'perror' in our example). This transition leads to the 'error_state'. Unspecified receptions are receptions which are not recognized at the present state of the protocol. ITSG generates unspecified receptions with the function 'ur'.

(parameter errors)

Generating testcases to test defensive behavior as for unspecified receptions but the reception is recognized as valid to a certain point in its syntax but then fails due to a parameter error such as a value error or a format error. ITSG generates both types of parameter errors with the function 'pe'. Note that time_outs may be treated as an unspecified reception ('ur') or a parameter error ('pe') depending on which way the logic specification specifies. If time_outs are not considered an error then the specification is complete with respect to the timers.

4.3.2. Typicals

Most of the connection establishment phase is covered with 20 loopless paths from 'idle' to 'final' which have been generated from the logic specification by executing the function 'auto_fire'. These loopless paths are given in Appendix B under the subtitle 'connection_establishment_paths'.

To generate the paths which test the transitions from 'data_transfer' to 'data_transfer' 'fire_one' is used. To determine which paths and combinations that should be generated the following heuristic is followed:

There are 4 cycle transitions (self loops) at the data_transfer state, namely, p13, p14, p15, and p16.

Lets call them a,b,c,d. Group them in groups of 1, then 2, and quit at groups of size 3 taking a representative set. There are 4 of size 1, i.e. a,b,c,d; 16 of size 2, namely aa,ab,ac,ad,ba,bb,bc,bd,ca,cb,cc,cd,da,db,dc,dd; and take a representative set of 4 of size 3, namely aaa,bbb,ccc,ddd. The total paths is 4 + 16 + 4 = 24. For each of these 24 cycle representatives there are 4 subpaths from 'idle' to 'data_transfer' and 3 subpaths from 'data_transfer' to 'final'. The subpaths (p1,p6), (p1,p7), (p3,p10) and (p4,p10) were chosen to be prefixes and the subpaths p17, p18, p19 were chosen as the postfixes. These subpaths were arbitrarily appended to the 24 cycle cases causing 6 repeats of the 4 prefix subpaths and 8 repeats of the 3 postfix subpaths. The resulting 24 paths are:

```
path 1:  p1,p5  --  p13 --  p17.
     2:  p1,p7  --  p14 --  p18.
     3:  p3,p10 --  p15 --  p19.
     4:  p4,p10 --  p16 --  p17.
     5:  p1,p6  --  p13,p13  -- p18.
     6:  p1,p7  --  p13,p14  -- p19.
     7:  p3,p10 --  p13,p15  -- p17.
*    8:  p4,p10 --  p13,p16  -- p18.
*    9:  p1,p6  --  p14,p13  -- p19.
*   10:  p1,p7  --  p14,p14  -- p17.
*   11:  p3,p10 --  p14,p15  -- p18.
*   12:  p4,p10 --  p14,p16  -- p19.
    13:  p1,p6  --  p15,p13  -- p17.
*   14:  p1,p7  --  p15,p14  -- p18.
    15:  p3,p10 --  p15,p15  -- p19.
    16:  p4,p10 --  p15,p16  -- p17.
*   17:  p1,p6  --  p16,p13  -- p18.
*   18:  p1,p7  --  p16,p14  -- p19.
*   19:  p3,p10 --  p16,p15  -- p17.
*   20:  p4,p10 --  p16,p16  -- p18.
    21:  p1,p6  --  p13,p13,p13  -- p19.
*   22:  p1,p7  --  p14,p14,p14  -- p17.
    23:  p3,p10 --  p15,p15,p15  -- p18.
```

* 24: p4,p10 -- p16,p16,p16 -- p19.

The dashes are only to differentiate between pre_paths, data_transfer cycles, and post_paths. Pre_paths and post_paths (given in Appendix B) can be generated using 'auto_fire'.

Note that ITSG will tell if these paths are not allowed when we try to generate the testcases corresponding to them. This protocol is simple and so it was not necessary to rely completely on ITSG. We have placed asterisks in front of these paths. It is clear that certain combinations are not feasible - such as p14 without a previous p13 i.e. outputting data to the N-1 service before receiving data from the N_service user first; or p16 without a previous p15 i.e. outputting to the N_service user without receiving data first from the N-1 service.

Thus, the number of potential paths for the data_transfer phase is 24. However, 12 of these paths are not feasible. The total correct and typical valued testcases for the connection phase and the data_transfer phase amounts to 20 + 12 = 32 altogether.

4.3.3. Atypicals

For atypicals, one can take a complete list of all input interactions and use them up in traversing paths from 'idle' to 'final' state. We identified 4 paths from 'idle' to 'final' to exercise all input interactions. They are:

1 p1,p8 to cover tcreq, dr

2 p1,p6,p13,p18 to cover cc,tdatr,ndind

3 p3,p10,p15,p17 to cover cr,tcres,dt,tdreq

4 p3,p10,p19 to cover nrind

For each interaction, we pick a path with that interaction and vary that interaction over the atypicals for every parameter value. Every change of typical to atypical is a new testcase. This could be done for parameter formats if the protocol had variable parameter formats.

The total number of testcases is 3 * 10 * 10 = 300 where the 3 is for high, low and middle atypicals and the 10 represents the number of input interactions and there exists an average of 10 values to vary per input interaction.

4.3.4. Unspecified Receptions

For the unspecified receptions, (and for parameter errors which are discussed in the following section) we must use paths with an error (i.e., 'perror' transition). We find such paths by executing 'auto_fire' from 'idle' to 'error_state'. The 'error_state' is reached by firing a special generic transition which can originate from any state (except 'final' and 'error_state'). This transition is represented by a clause which is of the following form.

```
t(perror, ANY_STATE, error_state,
        ERROR_STIMULUS, ['you are in the error state'])
                :-
                  input(ERROR_STIMULUS).
```

This transition clause is not part of the given logic specification but it is part of ITSG's processing facility (see Appendix D). ERROR_STIMULUS is generated by 'ur' or 'pe' functions only. The logic which controls their correct employment is that 'mak_tc' allows their use at 'break's which are inserted only and automatically in front of the error transition 'perror' (for error paths). Both functions check that the path is an error path or they do not succeed. (Of course 'atyp' checks that the path is not an error path).

'lur' can be employed at a 'break' and 'mak_tc' inserts a 'break' automatically in front of 'perror'. 'lur' generates a list of unspecified receptions for the present state (which is the state prior to firing 'perror'). This list of unspecified receptions are receptions in the logic specification which are currently not valid at the present state. The input alphabet 'I' consists of 10 symbols (input interaction identifiers) namely, cr, cc, dr, dt, ndind, nrind, tcreq, tcres, tdatr, and tdreq.

The list of unspecified receptions at state:

```
idle          is I - { tcreq, cr }       = 8   unspecified reception
wait_for_cc   is I - { dr, cc }          = 8          "
wait_for_tr   is I - { tcres, tdreq }    = 8          "
data_transfer is I - { tdatr, dt, tdreq,
                       ndind, nrind }    = 5          "
```

Hence, the total number of unspecified receptions for this

example is 29. Out of 29, we chose four examples which we give in Appendix C.

4.3.5. Parameter Errors

For parameter errors we use input interactions as we did for the atypicals. From the 8 error paths generated between 'idle' and 'error_state' by 'auto_fire' (we did not use 'fire_one') we must select some paths that allow us to cover all 10 different input interaction types. We collect a list of the 10 input interaction types and begin with the error paths (shortest ones), find the specified receptions for path (1), then for path (2), etc. until all input interactions are employed. The paths were taken from the transport_paths file. The format of the following table is path identification, actual path, executable receptions at the error point. We get:

error_path(1)	perror	[tsap,tcreq] [nsap,cr]
error_path(2)	p1,perror	[nsap,cc], [nsap,dr]
error_path(3)	p1,p6,perror	[tsap,tdatr] [nsap,dt] [tsap,tdreq] [nsap,ndind] [nsap,nrind]
error_path(5)	p3,perror	[tsap,tdatr]

We can put a parameter error in each input interaction field, i.e., 2 * 10 * 10 = 200, where the 2 represents invalid high and low and the 10's are the same as above. But as well as errors in the values, we can also generate

parameter format errors. To cover this, if we estimate each input interaction has 6 parameter fields on the average then we can generate testcases by varying and swapping parameter fields in various ways. For simplicity let us assume we systematically omit each field for each input interaction. Then, we get 10 interactions * 6 parameter fields each yielding 60 testcases.

Thus we get 32 typicals + 29 unspecified receptions + 300 atypicals + 260 parameter errors which yields 621 test-cases.

This simple CTS building scenario is aimed to show ITSG can easily and reliably generate an example from each of the cases mentioned above. ITSG is also tester friendly by providing an automatic test-purpose header for each testcase in addition to providing information, speed and consistency checking to help the tester make the test suite. For instance, a list of all input interactions in the protocol is provided simply by typing 'h(input)'.

In a less simple protocol, exhaustive techniques employed here must be abandoned in favour of representative cases. ITSG has been applied to such a protocol, namely the ILL protocol of the NLC (National Library of Canada) with success. Our conclusion is that ITSG becomes increasing valuable when the protocol becomes increasingly intractable. In the following section we discuss the examples which were generated for illustration.

4.4. Generation Of Test Cases

Having understood how ITSG generates the various typicals and anomalies via automatic default, 'atyp', 'ur' and 'pe', respectively we may wish to generate testcases corresponding to some paths constructed previously. Some paths correspond to testcases testing correct behavior and end at the state called 'final' and others are for testing defensive behavior and end at the state 'error_state'. Both types of paths begin at the initial state which is 'idle'.

All paths are passed to 'mak_tc' to generate the corresponding testcases. The resulting testcases are stored in external files. In the following subsection, we present the format of the testcase files.

4.4.1. Format of Testcases

There is one testcase per file and the filename is the timestamp of its creation. This facilitates automatic sorting and numbering of testcases. The 'testcase_template' is given at the comment section of each testcase so that a reader might know which execution path it is derived from (or it corresponds to).

Each testcase follows an overall testcase format which consists of the following.

(1) header

The header is composed of 3 divisions - a path, a trace, and a prolog clause representing the testcase.

(a) path

The path is a list of transitions constructed by using 'auto_fire' or 'fire_one'.

(b) trace

The trace illustrates:

(1) What anomaly was chosen.

(2) All changes that were made on the default values. This is non-existent for typicals of course. The changes can be very simple or become increasing complex. In any case, all that the tester does is traced and written. This is easily deciphered as the test-purpose at future times. Truly complicated things can always be further enhanced with comments.

(c) Prolog clause

The prolog clause is the origin of the testcase below the header. To produce the testcase this term has to be pruned and formatted by the 'mak_tc' function.

(2) testcase

There are 2 available output formats for the testcase. They are 'pretty print' and a new-line writing function. Examples of these output types are given in the following subsection.

4.4.2. Example Test Cases

We give 16 example TCs in Appendix C. The following is a breakdown of the sub-directory names with the number of testcases in each of these sub-directories.

tc_1_connection	4
tc_2_data_transfer	1
tc_3_ur	4
tc_4_atyp	4
tc_5_pe	3.

Testcases 'mar_20_932_pm' from the connection establishment group and 'mar_27_647_pm' from the unspecified receptions group in Appendix C are printed in the new_line format and the remainder of the testcases are in 'pretty-print' format. We choose two testcases from those given in Appendix C and present them here with annotations in order to provide examples of testcase formats.

In both testcases below, section 1 gives the filename of the testcase with the current page. Section 2 states the path from which the testcase was generated. The 'ur' example has an error in the path (via firing transition 'perror') and so a 'break' was automatically inserted in front of

'perror'. 'pe' or 'ur' must be generated at the error and 'atyp' or default must not be. For the 'atyp' example it happens we selected to put the 'break' after the first transition also.

Section 3 is filled with clauses that have the functor 'extensions'. This functor holds the trace made from the user's actions when s/he prepared the anomaly. In 'ur' the first 'extensions' tells an 'ur' anomaly type was selected and that the actual anomaly was '[tsap,tcres]', i.e., after a transport connection request ([tsap,tcreq]), it is not specified that a transport connection response ([tsap,tcres]) can occur. For this section in the 'atyp' example, the functor 'extensions' had the value 'atyp' because the user was making an atypical anomaly. The second 'extensions' functor in 'ur' tells that 'input' was given the unspecified reception and the actual values are displayed. However the tester only typed 'ur([tsap,tcres])'. For the atypical example the 'extensions' show that the field 'source_ref' was given the value '65535' which is the valid maximum. This is obvious and simple and no comments were left remarking on the value. In both examples, section 4 is the Prolog clause of the testcase.

At this point the testcase heading is completed. Section 5 presents the formatted testcase. 'ur' presents the testcase in the more compact, less friendly newline-write

format and 'atyp' presents the testcase in the pretty-print format.

Section 6 is the comment section although comments can go anywhere as exemplified in the atypical example which has a comment mid-way through the pretty-printed testcase. This section presents the testcase_template which is a template that must be filled for 'mak_tc' to succeed.

Example 1: New Line Format

```
1/ example_TCs/tc_3_ur/mar_27_647_pm Page 1

2/ path([pl,break,perror]).

3/ extensions(ur([tsap,tcres])).

   extensions(input([tsap,tcres,[82,0,normal,'[]']]),true).

4/ path_value(0,[
   [[[tsap,tcreq,[213,44,487,1,normal,[]]],[]],[[],[nsap,cr,
[1,414,[],0,normal,487,44,128]]]],
   [[[tsap,tcres,[82,0,normal,[]]]],[[you are,in the,error
state]]]]).

5/
[[tsap,tcreq,[213,44,487,1,normal,[]]],[nsap,cr,[1,414,[],0,
normal,487,44,128]]]
   [[tsap,tcres,[82,0,normal,[]]],[you are,in the,error
state]]

6/ Comment
   -------

   the tc_template is:

   1 : input_file    : transport_paths
   2 : input_functor : error_path
   3 : input_number  : 2
   4 : output_file   : mar_27_647_pm
   5 : output_type   : nl_write
```

Example 2: Pretty Print Format

```
1/ example_TCs/tc_4_atyp/mar_29_1201_pm Page 1

2/ path([pl,break,p8]).
```

```
3/ extensions(atyp).

   extensions(source_ref(65535),true).

4/ path_value(0,[
   [[[tsap,tcreq,[165,64,980,1,normal,[]]],[]],[[],[nsap,cr,
[1,1179,[],0,normal,980,64,128]]]],
   [[[],[nsap,dr,[1179,65535,[data],1]]],[[tsap,tdind,[165,
ts_u_nrm,[data]]],[nsap,ndreq,[1]]]]
   ]).

5/      tsap
        tcreq
            165
            64
            980
            1
            normal
            []

        nsap
        cr
            1
            1179
            []
            0
            normal
            980
            64
            128

        nsap
        dr
            1179
            65535              (* the atypical we wanted *)
              data

            1


        tsap
        tdind
            165
            ts_u_nrm
              data

        nsap
        ndreq
            1
```

6/ Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 7
4 : output_file    : mar_29_1201_pm
5 : output_type    : pp
```

5. Conclusions

In this thesis, we have presented an interactive testsequence generator, called ITSG, which provides an interface between a tester and a logic specification. The basic functionality of this interactive tool is to help the tester constructing testcases to test both the correct functionality and the defensive behavior of an implementation under test. Thus, besides providing a controlled traversal of the state-space of the given logic specification (to generate tests for testing correct functionality of the implementation under test), ITSG provides functions to generate erroneous and exceptional stimuli to the implementation under test.

In order to demonstrate the feasibility of the interactive testsequence generator, we have illustrated the ease of use and the flexibility of this interactive testsequence generator on the construction of an example conformance testsuite for OSI Transport protocol class 0 implementations.

ITSG is fast and friendly. Its throughput is very high. (I mean to say that miniscule tester action can generate extreemely large testcases because the tester only specifies the error and the error location and then ITSG fills the testcase automatically.) Relevant information can be accessed spontaneously and whenever needed. The information can be interactively used to generate testcases which can be

manufactured in seconds. Further the testcases are accurate, in a user friendly format and automatically provided with a testcase header which contains the path, the anomaly and how the anomaly was generated i.e. test purpose. As well, ITSG has algorithmic speed enhancements. The auto-fire traversal procedure was speed improved and ITSG releases the extensions part of the specification unless it is needed thereby improving significantly the execution speed of many ITSG functions.

ITSG is flexible. It allows a protocol status information file to be consulted upon startup and the options of the present status to be easily toggled on or off at any time. This allows all aspects of the protocol to be simulated by a tester with perfect ease. Also because the protocol (i.e. logic specification) is primarily a list of prolog clauses it is very easy to partition. The transitions list (automaton) can be adapted to any possible PICS allowed by the protocol. Similarly for the extensions - alternate format specifications bases and alternate values bases may be created at specific files. Different files can be loaded with ITSG thereby allowing for variable 'default' or 'typical' conditions in the CTS generation.

Format of the resulting testcases can be adapted to TTCN (Tree and Tabular Combined Notation). This adaptation can be based on the automated testing feature of ITSG (i.e., testing driven by a CTS residing on a file). Testcases in

TTCN could then be generated with ITSG by specifying a collection of subtrees, each representing a collection of subpaths. From these subpaths, ITSG generates testcases in TTCN format. Finally, the PDU and service primitive parameter descriptions in TTCN can be generated from the extensions part of the LS by ITSG.

Future developments of ITSG might be to remove the tester. Foundational matters upon which an automated system must rest are completed. Anomaly generation could be automated. Research might suggest particularly useful enumerations or algorithms to drive the selection of anomalies or perhaps even a set of replacable heuristics to decide selections as an Expert System might do. Similarly the selection of paths can be effected by adding intelligence to the 'auto_fire' function. Decisions such as dynamic traversal and pruning of possibilities could be effected by sets of heuristics thereby improving 'auto_fire's current loopless and path_counting_halt static capability.

6. REFERENCES

[Boch 80] G.v. Bochmann, "A general transition model for protocols and communication services", IEEE Trans. on Communications, Vol.28, No.4, 1980, pp. 643-650.

[ClMe 81] W.F. Clocksin and C.S. Mellish, Programming in Prolog, Springer-Verlag, 1981.

[ISO 82] G.v. Bochmann, "Examples of transport protocol specifications", ISO TC/97/SG16/WG1, Subgroup B on FDT, Nov. 1982.

[ISO 85a] Draft Proposal of Estelle, A FDT based on an extended state transition model, ISO TC97/SC21 DP 9074, June 4, 1985.

[ISO 85b] Revised Working Draft for OSI Conformance Testing Methodology and Framework, ISO TC97/SC21 N909, Project Number 97.21.23, Nov 1985.

[LiMc 83] R.J. Linn and W.H. McCoy, "Producing tests for implementations of OSI protocols", Proc. of 3rd IFIP/WG6.1 Int. Workshop on Protocol Specification, Verification and Testing, Zurich, May 1983, pp. 505-520.

[MiHo 81] Edward Miller and William E. Howden, Tutorial: Software Testing and Validation, 2nd edition, IEEE Computer Society Press, 1981.

[Myer 79] Glenford J. Myers, The Art of Software Testing, John Wiley-Interscience Publication, 1979.

[Piat 80] T.F. Piatkowski, "Remarks on the feasibility of validating and testing ADCCP implementations", Proc. of Trends and Applications Symposium (NBS), 1980, Gaithersburg, MD.

[Rayn 82] D. Rayner, " A system for testing protocol implementations", NPL Report DITC 9/82, August 1982, (also in "Protocol Specification, Testing, and Verification" (ed. C. Sunshine), North-Holland, 1982 pp. 539-553).

[Rayn 85] D. Rayner, "Standardizing conformance testing for OSI", Proc. of 5th IFIP/WG6.1 Int. Workshop on Protocol Specification, Verification, and Testing, Toulouse, France, June 1985, pp. 10.1-10.22

[SaBo 84] B. Sarikaya and G.v. Bochmann, "Synchronization and specification issues in protocol testing", IEEE Trans. on Communications, Vol.32, No.4, 1984, pp. 389-395.

[SaBo 85] B. Sarikaya and G.v. Bochmann, "Obtaining normal form specifications for protocols", Proc. of COMNET'85, Budapest, Hungary, Oct 1985, pp. 6.133-6.149.

[SBC 85] B. Sarikaya, G.v. Bochmann, and E. Cerney, "A test design methodology for protocol testing", Proc. of 18th Hawaii ICSS, Jan 1985, Vol.2, pp. 710-721.

[Sidh 83] D.P. Sidhu, "Protocol verification via executable logic specifica tions", Proc. of 3rd IFIP/WG6.1 Int. Workshop on Protocol Specification, Verification and Testing, Zurich, May 1983, pp. 237-248.

[Tane 81] A. S. Tanenbaum, "Computer Networks", Prentice-Hall, INC., Englewood Cliffs, New Jersey 07632.

[Ural 85] H. Ural, "Construction of logic specifications for communications protocols and services", Tech. Report : TR-85-17, Dept. of CSI, Univ. of Ottawa, 1985.

[Ural 86] H. Ural, "Logic specifications for communication systems", Proc. of PCCC-86, Phoenix, Arizona, March 1986.

[UrPr 83] H. Ural and R.L. Probert, "User-guided test sequence generation", Proc. of 3rd IFIP/WG6.1 Int. Workshop on Protocol Specification, Verification and Testing, Zurich, May 1983, pp. 421-436.

[UrPr 84] H. Ural and R.L. Probert, "Automated testing of Protocol Specifications and their implementations", Proc. of ACM SIGCOMM 1984, Montreal, June 1984, pp. 149-155.

[UrPr 86] H. Ural and R.L. Probert, "Step-wise validation of communication protocols and services", Computer Networks, Vol. 11, No. 4, PP. 183-202.

[Zimm 80] H. Zimmerman, "OSI Reference Model - The ISO Model of Architecture for Open Systems Connection", IEEE Transactions on Communications, Vol. COMM-28, No. 4, April 1980.

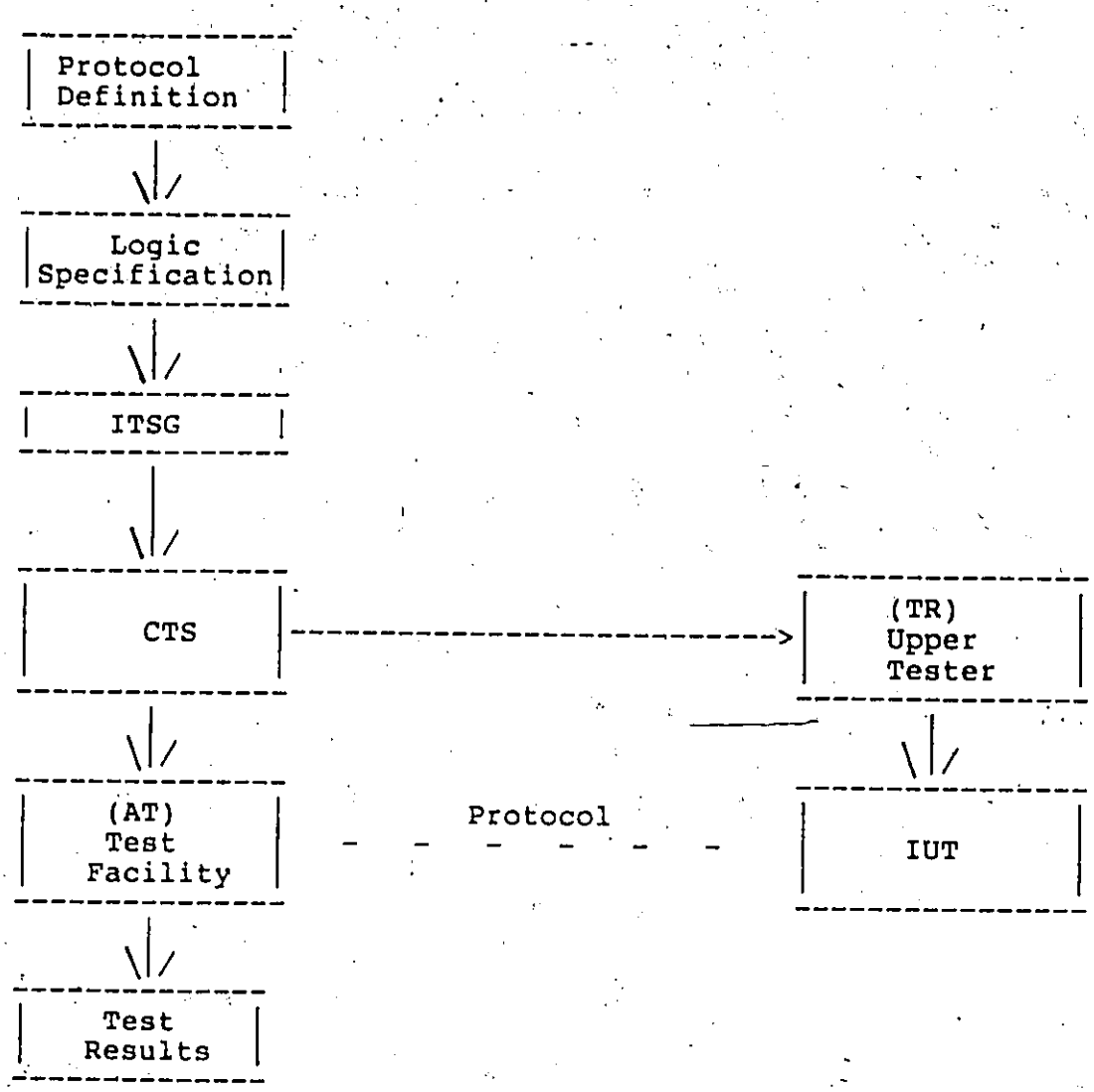

FIGURE 1. A Test Facility Model With ITSG

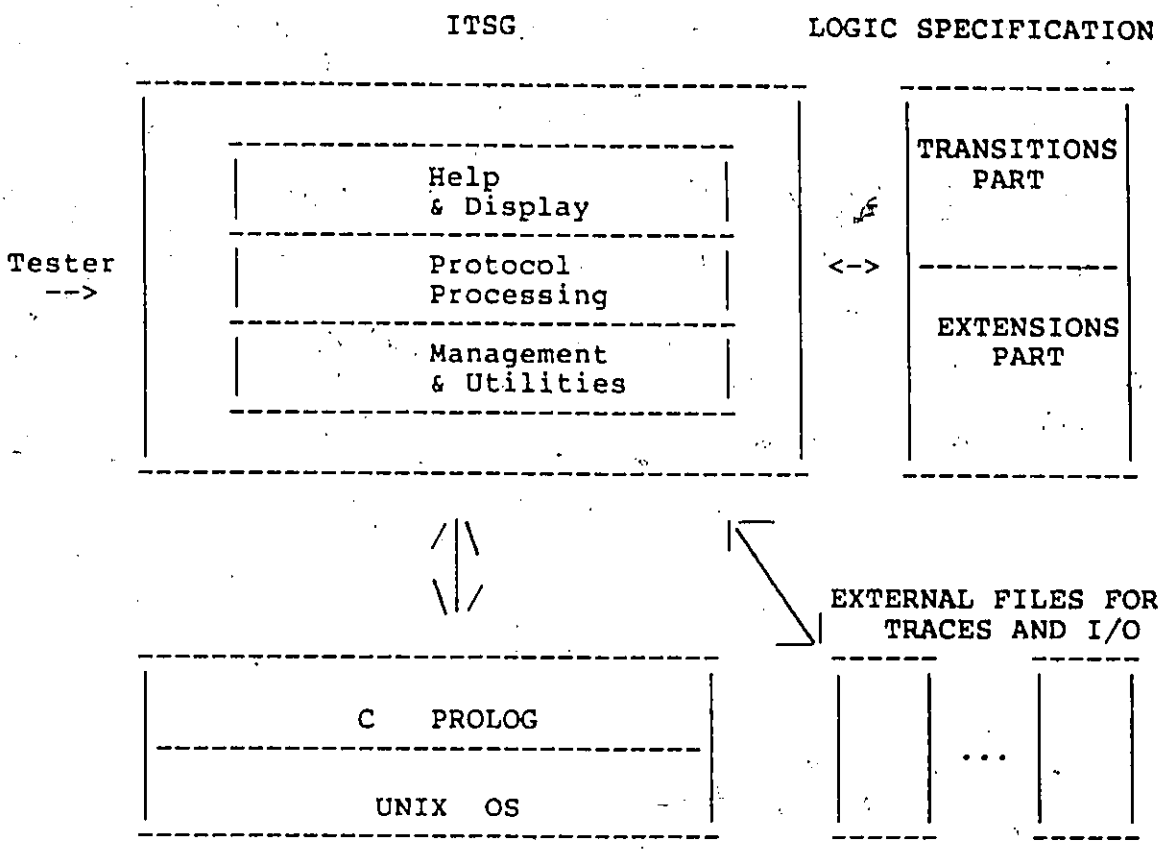

FIGURE 2. ITSG (INTERACTIVE TESTSEQUENCE GENERATOR)

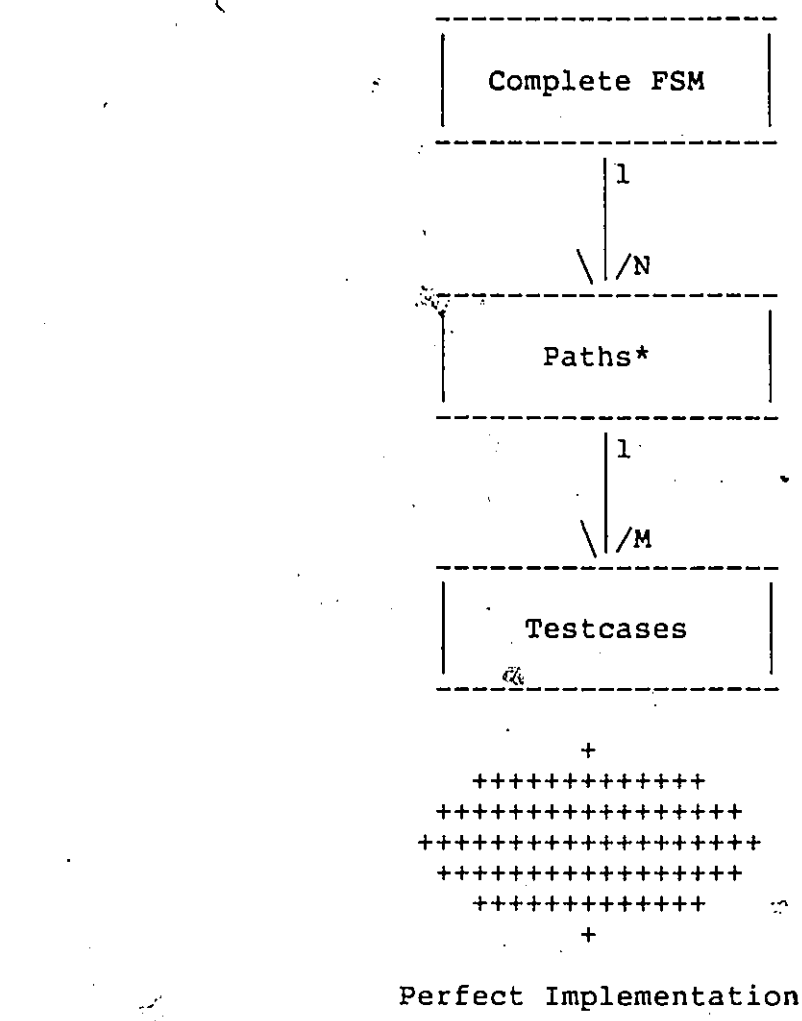

FIGURE 3. ITSG's Methodology Reference Model

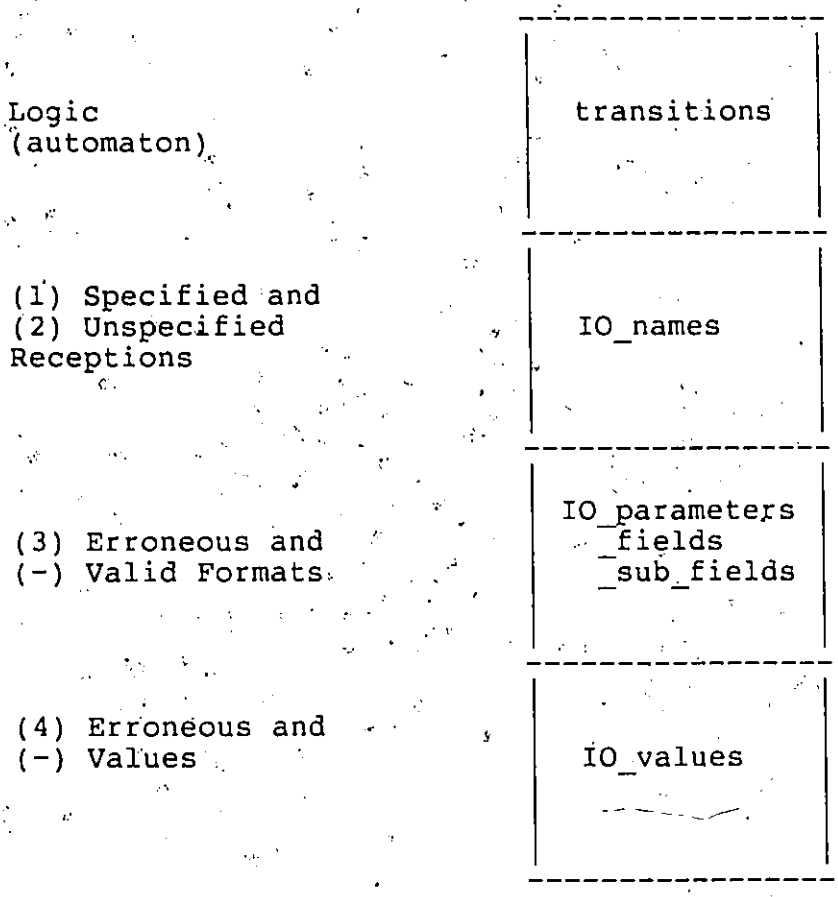

FIGURE 4. A Reference Model of Logic Specifications

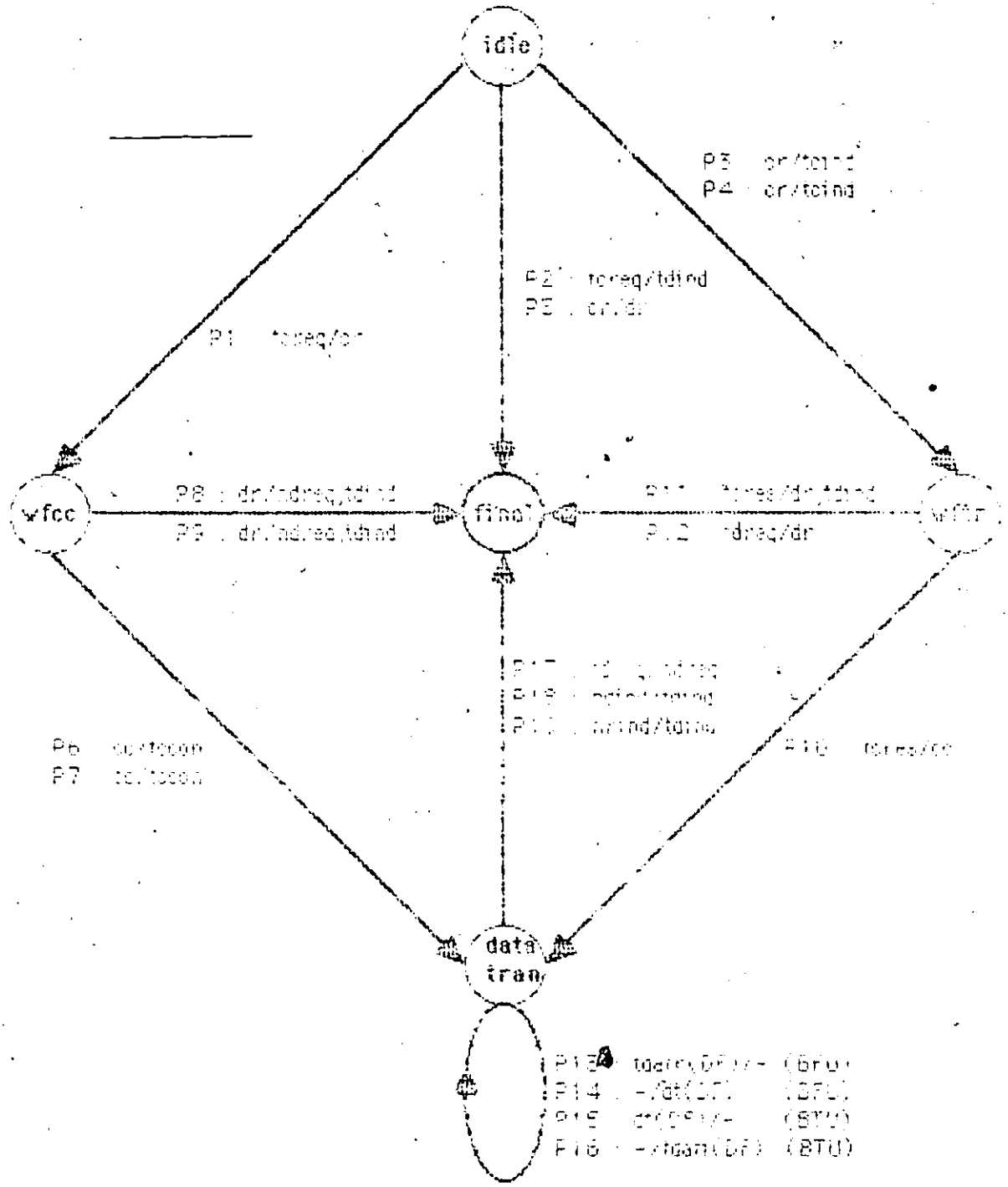

FIGURE 5. Transitions Part of Logic Specification (TP Class 0)

APPENDICES

APPENDIX A

LOGIC SPECIFICATION FOR THE TRANSPORT PROTOCOL CLASS 0

```
t(p1,
  [idle,           PTCEP, PTPDU_S, PQOS, BFU, BTU, SSEQ, RSEQ,
   PLREF, RREF, PCGTA, PCDTA],
  [wait_for_cc,    NTCEP, NTPDU_S, NQOS, BFU, BTU, SSEQ, RSEQ,
   NLREF, RREF, NCGTA, NCDTA],
  [ [tsap, tcreq, IPL] , []], [[], [nsap, cr   , OPL]]) :-
  /* acceptable_QOS */
  IPL = [NTCEP, NCDTA, NCGTA, NQOS, 'normal', TSDATA],
  input([tsap, tcreq, IPL]),
  OPL = [R1, NLREF, TSDATA, 0, 'normal', NCGTA, NCDTA, NTPDU_S],
  output([nsap, cr   , OPL]),
  (var(NTPDU_S) ->  true   ; NTPDU_S>0).        /* either/or */

t(p2,
  [idle,           TCEP, TPDU_S, PQOS, BFU, BTU, SSEQ, RSEQ,
   LREF, RREF, PCGTA, PCDTA],
  [final,          TCEP, TPDU_S, NQOS, BFU, BTU, SSEQ, RSEQ,
   LREF, RREF, NCGTA, NCDTA],
  [[tsap, tcreq,IPL] , []], [[tsap, tdind,OPL] , []]) :-
  /*  unacceptable QOS  */
  IPL = [TCEP, NCDTA, NCGTA, NQOS, 'expedited', TSDATA],
  input([tsap, tcreq,IPL]),
  OPL = [TCEP, 'ts_qual_fail', P3],
  output([tsap, tdind,OPL]).

t(p3,
  [idle,           PTCEP, PTPDU_S, PQOS, BFU, BTU, SSEQ, RSEQ,
   LREF, PRREF, PCGTA, PCDTA],
  [wait_for_tcres, NTCEP, NTPDU_S, NQOS, BFU, BTU, SSEQ, RSEQ,
   LREF, NRREF, NCGTA, NCDTA],
  [[], [nsap, cr   ,IPL] ], [[tsap, tcind,OPL], []]) :-
  /*  acceptable_QOS
      tpdu_size_defined  */
  IPL = [P1, NRREF, TSDATA, 0, 'normal', NCGTA, NCDTA, NTPDU_S],
  input([nsap, cr   ,IPL] ),
  (var(NTPDU_S) -> true ; (NTPDU_S>0,NTPDU_S < 2049)),
  /* either/or */
  OPL = [NTCEP, NCDTA, NCGTA, NQOS, 'normal', TSDATA],
  output([tsap, tcind,OPL] ).
```

```
t(p4,
  [idle,           PTCEP, PTPDU_S, PQOS, BFU, BTU, SSEQ, RSEQ,
   LREF, PRREF, PCGTA, PCDTA],
  [wait_for_tcres, NTCEP, 128    , NQOS, BFU, BTU, SSEQ, RSEQ,
   LREF, NRREF, NCGTA, NCDTA],
  [[], [nsap, cr   ,IPL] ], [[tsap, tcind,OPL] , []]) :-
  /*  acceptable_QOS
      tpdu_size_undefined  */
  IPL = [P1, NRREF, TSDATA, 0, 'normal', NCGTA, NCDTA, TPDU_S],
  input([nsap, cr   ,IPL] ),
  TPDU_S = 0,
  OPL = [NTCEP, NCDTA, NCGTA, NQOS, 'normal', TSDATA],
  output([tsap, tcind,OPL] ).

t(p5,
  [idle,           TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ,
   PLREF, PRREF, PCGTA, PCDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ,
   NLREF, NRREF, NCGTA, NCDTA],
  [[], [nsap, cr   ,IPL] ], [[], [nsap, dr   ,OPL] ]) :-
  /*  unacceptable_QOS  */
  IPL = [P1, NRREF, TSDATA, 0,'expedited', NCGTA, NCDTA, TPDU_S],
  input([nsap, cr   ,IPL] ),
  OPL = [NRREF, NLREF, TSDATA, 3],
  output([nsap, dr   ,OPL] ).

t(p6,
  [wait_for_cc,    TCEP, PTPDU_S, QOS, PBFU, PBTU, PSSEQ, PRSEQ,
   LREF, PRREF, CGTA, CDTA],
  [data_transfer,  TCEP, NTPDU_S, QOS, [] , [] , 0   , 0   ,
   LREF, NRREF, CGTA, CDTA],
  [[], [nsap, cc   ,IPL] ], [[tsap, tccon,OPL] , []]) :-
  /*  tpdu_size_defined  */
  IPL=[P1, LREF, NRREF, TSDATA, 0,'normal',CGTA, CDTA, NTPDU_S],
  input([nsap, cc   ,IPL] ),
  ( var(NTPDU_S) -> true ;  (NTPDU_S > 0, NTPDU_S < 2049)  ),
  /* either/or */
  ( var(NTPDU_S) -> true ;  not(NTPDU_S > PTPDU_S) ),
  /* either/or */
  OPL = [TCEP, QOS, 'normal', TSDATA],
  output([tsap, tccon,OPL] ).

t(p7,
  [wait_for_cc,    TCEP, PTPDU_S, QOS, PBFU, PBTU, PSSEQ, PRSEQ,
   LREF, PRREF, CGTA, CDTA],
  [data_transfer,  TCEP, 128   , QOS, [] , [] , 0   , 0   ,
   LREF, NRREF, CGTA, CDTA],
  [[], [nsap, cc   ,IPL] ], [[tsap, tccon,OPL] , []]) :-
  /*  tpdu_size_undefined  */
  IPL=[P1, LREF, NRREF, TSDATA, 0, 'normal',CGTA, CDTA, TPDU_S],
  input([nsap, cc   ,IPL] ),
  TPDU_S = 0,
  OPL = [TCEP, QOS, 'normal', TSDATA],
  output([tsap, tccon,OPL] ).
```

```
t(p8,
  [wait_for_cc,    TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   PRREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   NRREF, CGTA, CDTA],
  [[],[nsap,dr,IPL]], [[tsap,tdind,OPL1],[nsap,ndreq,OPL2]]) :-
  /* peer_user_initiated_termination */
  IPL = [LREF, NRREF, USDATA, 1],
  input([nsap, dr    ,IPL] ),
  OPL1 = [TCEP, 'ts_u_nrm', USDATA],
  output([tsap, tdind,OPL1] ),
  OPL2 = [NCEP],
  output([nsap, ndreq,OPL2] ).

t(p9,
  [wait_for_cc,    TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   PRREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   NRREF, CGTA, CDTA],
  [[],[nsap,dr,IPL]], [[tsap,tdind,OPL1],[nsap,ndreq,OPL2]]) :-
  /* not_peer_user_initiated_termination */
  IPL = [LREF, NRREF, USDATA, P],
  input([nsap, dr    ,IPL] ),
  ( var(P) -> true ; not(P = 1) ),     /* either/or */
  OPL1 = [TCEP, R2, USDATA],
  output([tsap, tdind,OPL1] ),
  ( var(R2) -> true ; not(R2='ts_u_nrm') ),    /* either/or */
  OPL2 = [NCEP],
  output([nsap, ndreq,OPL2] ).

t(p10,
  [wait_for_tcres, TCEP, PTPDU_S, PQOS,  PBFU, PBTU, PSSEQ,
   PRSEQ, PLREF, RREF, CGTA, CDTA],
  [data_transfer, TCEP, NTPDU_S, NQOS,  []  , []  , 0   ,
   0   , NLREF, RREF, CGTA, CDTA],
  [[tsap, tcres,IPL], []], [[], [nsap, cc    ,OPL]]) :-
  /*  requested_QOS_LE_proposed_QOS */
  IPL = [TCEP, NQOS, 'normal', TSDATA],
  input([tsap, tcres,IPL] ),
  (var(NQOS) -> true ; NQOS =< PQOS),     /* either/or */
  OPL = [P1,RREF,NLREF,TSDATA,P5,'normal',CGTA,CDTA,NTPDU_S],
  ( var(NTPDU_S)->true;not(NTPDU_S>PTPDU_S) ),/* either/or */
  output([nsap, cc    ,OPL] ).
```

```
t(pl1,
  [wait_for_tcres, TCEP, TPDU_S, PQOS, BFU, BTU, SSEQ, RSEQ,
   PLREF, RREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, NQOS, BFU, BTU, SSEQ, RSEQ,
   NLREF, RREF, CGTA, CDTA],
  [[tsap,tcres,IPL],[]], [[tsap,tdind,OPL1],[nsap,dr,OPL2]]) :-
  /* requested_QOS_GT_proposed_QOS */
  IPL = [TCEP, NQOS, 'normal', TSDATA],
  input([tsap, tcres,IPL]),
  ( var(NQOS) -> true ; NQOS > PQOS ),     /* either/or */
  OPL1 = [TCEP, 'ts_qual_fail', []],
  output([tsap, tdind,OPL1] ),
  OPL2 = [RREF, NLREF, TSDATA, 3],
  output([nsap, dr   ,OPL2] ).

t(pl2,
  [wait_for_tcres, TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ,
   PLREF, RREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ,
   NLREF, RREF, CGTA, CDTA],
  [[tsap, tdreq,IPL], []], [[], [nsap, dr   , OPL]]) :-
  /*  user_initiated_termination  */
  IPL = [RI, USER],
  input([tsap, tdreq,IPL] ),
  OPL = [RREF, NLREF, USER, 1],
  output([nsap, dr   , OPL]).

t(pl3,
  [data_transfer,  TCEP, TPDU_S, QOS, PBFU, BTU, SSEQ, RSEQ,
   LREF, RREF, CGTA, CDTA],
  [data_transfer,  TCEP, TPDU_S, QOS, NBFU, BTU, SSEQ, RSEQ,
   LREF, RREF, CGTA, CDTA],
  [[tsap, tdatr,IPL] , []], [[], []]) :-
  /* flow_control_from_user_is_ready */
  IPL = [P1, DF, P3],
  input([tsap, tdatr,IPL]),
  insert(PBFU, DF, NBFU).

t(pl4,
  [data_transfer,  TCEP, TPDU_S, QOS, PBFU, BTU, PSSEQ, RSEQ,
   LREF, RREF, CGTA, CDTA],
  [data_transfer,  TCEP, TPDU_S, QOS, NBFU, BTU, NSSEQ, RSEQ,
   LREF, RREF, CGTA, CDTA],
  [[], []], [[], [nsap, dt   ,OPL]]) :-
  /* flow_control_to_provider_is_ready */
  not(PBFU = []),
  delete(PBFU, DF, NBFU),
  OPL = [RREF, LREF, DF, NSSEQ, EOF],
  NSSEQ is PSSEQ + 1,
  output([nsap, dt   ,OPL]).
```

```
t(pl5,
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, PBTU, SSEQ, PRSEQ,
   LREF, RREF, CGTA, CDTA],
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, NBTU, SSEQ, NRSEQ,
   LREF, RREF, CGTA, CDTA],
  [[], [nsap, dt   ,IPL]], [[], []]) :-
  /* flow_control_from_provider_is_ready */
  IPL = [LREF, RREF, DF, NRSEQ, EOF],
  NRSEQ is PRSEQ + 1,
  input([nsap, dt   ,IPL]),
  insert(PBTU, DF, NBTU).

t(pl6,
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, PBTU, SSEQ, RSEQ,
   LREF, RREF, CGTA, CDTA],
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, NBTU, SSEQ, RSEQ,
   LREF, RREF, CGTA, CDTA],
  [[], []], [[tsap, tdati,OPL], []]) :-
  /* flow_control_to_user_is_ready */
  not(PBTU = []),
  delete(PBTU, DF, NBTU),
  OPL = [P1, DF, P3],
  output([tsap, tdati,OPL]).

t(pl7,
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   RREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   RREF, CGTA, CDTA],
  [[tsap, tdreq,IPL], []], [[], [nsap, ndreq, OPL]]) :-
  /*  user_initiated_termination  */
  IPL = [TCEP, DISR],
  /*  BTU = [],
      BFU = [],  */
  input([tsap, tdreq,IPL]),
  output([nsap, ndreq, OPL]).

t(pl8,
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   RREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   RREF, CGTA, CDTA],
  [[], [nsap, ndind, IPL]], [[tsap, tdind, OPL], []]) :-
  /*  BTU = [],
      BFU = [],  */
  input([nsap, ndind, IPL]),
  OPL = [TCEP, 'ts fail', []],
  output([tsap, tdind, OPL]).
```

```
t(p19,
  [data_transfer,  TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   RREF, CGTA, CDTA],
  [final,          TCEP, TPDU_S, QOS, BFU, BTU, SSEQ, RSEQ, LREF,
   RREF, CGTA, CDTA],
  [[], [nsap, nrind, IPL]], [[tsap, tdind, OPL], []]) :-
  /*  BTU = [],
      BFU = [],  */
  input([nsap, nrind, IPL]),
  OPL = [TCEP, 'ts_fail', []],
  output([tsap, tdind, OPL]).

input([AP, I, LP]) :-
((I = tcreq, t_connect_req(LP))
;(I = tcres, t_connect_resp(LP))
;(I = tdatr, t_data_req(LP))
;(I = tdreq, t_disconnect_req(LP))
;(I = cr   , cr_TPDU(LP))
;(I = cc   , cc_TPDU(LP))
;(I = dt   , dt_TPDU(LP))
;(I = dr   , dr_TPDU(LP))
;(I = nrind, n_reset_ind(LP))
;(I = ndind, n_disconnect_ind(LP))).

output([AP, O, LP]) :-
((O = tcind, t_connect_ind(LP))
;(O = tccon, t_connect_conf(LP))
;(O = tdati, t_data_ind(LP))
;(O = tdind, t_disconnect_ind(LP))
;(O = cr   , cr_TPDU(LP))
;(O = cc   , cc_TPDU(LP))
;(O = dt   , dt_TPDU(LP))
;(O = dr   , dr_TPDU(LP))
;(O = ndreq, n_disconnect_req(LP))).
```

```
t_connect_req([TCEP, TTA, FTA, QOS, OPT, TSDATA]) :-
tcep(TCEP),
to_T_address(TTA),
from_T_address(FTA),
quality_of_TS(QOS),
options(OPT),
ts_data(TSDATA).

t_connect_ind([TCEP, TTA, FTA, QOS, OPT, TSDATA]) :-
tcep(TCEP),
to_T_address(TTA),
from_T_address(FTA),
quality_of_TS(QOS),
options(OPT),
ts_data(TSDATA).

t_connect_resp([TCEP, QOS, OPT, TSDATA]) :-
tcep(TCEP),
quality_of_TS(QOS),
options(OPT),
ts_data(TSDATA).

t_connect_conf([TCEP, QOS, OPT, TSDATA]) :-
tcep(TCEP),
quality_of_TS(QOS),
options(OPT),
ts_data(TSDATA).

t_disconnect_req([TCEP, USER]) :-
tcep(TCEP),
ts_user_reason(USER).

t_disconnect_ind([TCEP, DISR, USER]) :-
tcep(TCEP),
ts_disconnect_reason(DISR),
ts_user_reason(USER).

t_data_req([TCEP, UDATA, LF]) :-
tcep(TCEP),
ts_user_data(UDATA),
last_fragment_of_TSDU(LF).

t_data_ind([TCEP, UDATA, LF]) :-
tcep(TCEP),
ts_user_data(UDATA),
last_fragment_of_TSDU(LF).
```

```
tcep(TCEP) :- (nonvar(TCEP), (TCEP > 0, TCEP < 500))
              ;  (var(TCEP), random(TCEP, 500)).

to_T_address(TTA) :- (nonvar(TTA), true)
                     ;  (var(TTA), random(TTA,999)).

from_T_address(FTA) :- (nonvar(FTA), true)
                       ;  (var(FTA), random(FTA,999)).

ts_data(TSDATA) :- (nonvar(TSDATA), true)
                   ;  (var(TSDATA), TSDATA = []).

ts_disconnect_reason('ts_u_nrm').
ts_disconnect_reason('ts_cong').
ts_disconnect_reason('ts_fail').
ts_disconnect_reason('ts_qual_fail').
ts_disconnect_reason('u_unknown').


ts_user_reason(USER) :-
 (nonvar(USER), true); (var(USER), USER = 'reason').

options('normal').
options('expedited').

quality_of_TS(QOS) :- (nonvar(QOS), true)
                      ;   (var(QOS), (random(QOS,9); QOS = 10)).

ts_user_data(UDATA) :- ((nonvar(UDATA), true)
                       ;   (var(UDATA), maxudl(MAXUDL),
                            random(L,MAXUDL), fillud(UDATA, L))).

maxudl(100).

last_fragment_of_TSDU(LF) :- (nonvar(LF), (LF = 0; LF = 1))
                             ;   (var(LF), random(LF,1)).
```

```
cr_TPDU([CRD, SREF, TSDATA, CL, OPT, CGTA, CDTA, TPDU]) :-
        credit_value(CRD),
        source_ref(SREF),
        user_data(TSDATA),
        class_ind(CL),
        options_ind(OPT),
        calling_addr(CGTA),
        called_addr(CDTA),
        tpdu_size_ind(TPDU).

dr_TPDU([DREF, SREF, TSDATA, DISR]) :-
        dest_ref(DREF),
        source_ref(SREF),
        user_data(TSDATA),
        disconnect_reason(DISR).

cc_TPDU([CRD, DREF, SREF, TSDATA, CL, OPT, CGTA, CDTA, TPDU]) :-
        credit_value(CRD),
        dest_ref(DREF),
        source_ref(SREF),
        user_data(TSDATA),
        class_ind(CL),
        options_ind(OPT),
        calling_addr(CGTA),
        called_addr(CDTA),
        tpdu_size_ind(TPDU).

dt_TPDU([DREF, SREF, DATA, SEQ, EOF]) :-
        dest_ref(DREF),
        source_ref(SREF),
        user_data(DATA),
        sequence(SEQ),
        end_of_TSDU(EOF).

credit_value(CRD) :- (nonvar(CRD), true); (var(CRD), CRD = 1).

user_data(UD) :- (nonvar(UD), true); (var(UD), UD = [data]).

class_ind(0).

options_ind('normal').
options_ind('expedited').

calling_addr(CGTA) :- from_T_address(CGTA).
called_addr(CDTA) :- to_T_address(CDTA).
```

```
tpdu_size_ind(TPDU) :- (nonvar(TPDU), true)
                       ;  (var(TPDU), (TPDU = 0;
                                         TPDU = 128;
                                         TPDU = 256;
                                         TPDU = 512;
                                         TPDU = 1024;
                                         TPDU = 2048;
                                         TPDU = 2049)).

reference(REF) :- (nonvar(REF), REF > 0, REF < 65536)
                  ;  (var(REF), random(REF, 65536)).

source_ref(SREF) :- reference(SREF).

dest_ref(DREF) :- reference(DREF).

disconnect_reason(DISR) :- (nonvar(DISR), (DISR > 0, DISR < 6))
                           ;(var(DISR), random(DISR, 5)).

end_of_TSDU(EOF) :- last_fragment_of_TSDU(EOF).

sequence(_).

n_disconnect_req([NCEP]) :- ncep(NCEP).

n_disconnect_ind([NCEP, DISR]) :- ncep(NCEP),
                                  ns_disconnect_reason(DISR).

n_reset_ind([NCEP]) :- ncep(NCEP).

ncep(1).

ns_disconnect_reason('nz_xdc').
ns_disconnect_reason('nc_abo').


insert(X, Y, Z) :- append(X, [Y], Z).

delete([Y|Z], Y, Z).

seed(997).

random(X, UB) :- retract(seed(P)),
                 X is P mod (UB + 1),
                 N is (125 * P + 1) mod 4097,
                 assert(seed(N)), !.

fillud([], 0) :- !.
fillud([X|Y], L) :- random(X, 255),
                    LL is L - 1,
                    fillud(Y, LL).

initial_state([idle,_,_,_,_,_,_,_,_,_,_,_]).
final([final|R]).
```

APPENDIX B

EXAMPLE PATHS FROM THE EXAMPLE CTS FOR THE TRANSPORT PROTOCOL CLASS 0

```
transport paths
/* from idle -> final */

connection_establishment_path(1,[p1,p6,p17]).
connection_establishment_path(2,[p1,p6,p18]).
connection_establishment_path(3,[p1,p6,p19]).
connection_establishment_path(4,[p1,p7,p17]).
connection_establishment_path(5,[p1,p7,p18]).
connection_establishment_path(6,[p1,p7,p19]).
connection_establishment_path(7,[p1,p8]).
connection_establishment_path(8,[p1,p9]).
connection_establishment_path(9,[p2]).
connection_establishment_path(10,[p3,p10,p17]).
connection_establishment_path(11,[p3,p10,p18]).
connection_establishment_path(12,[p3,p10,p19]).
connection_establishment_path(13,[p3,p11]).
connection_establishment_path(14,[p3,p12]).
connection_establishment_path(15,[p4,p10,p17]).
connection_establishment_path(16,[p4,p10,p18]).
connection_establishment_path(17,[p4,p10,p19]).
connection_establishment_path(18,[p4,p11]).
connection_establishment_path(19,[p4,p12]).
connection_establishment_path(20,[p5]).


/* pre_paths to data_transfer */

/* from idle -> data_transfer */

pre_path(1,[p1,p6]).
pre_path(2,[p1,p7]).
pre_path(3,[p3,p10]).
pre_path(4,[p4,p10]).

/* post_paths from data_transfer */

/* from data_transfer -> final */

post_path(1,[p17]).
post_path(1,[p18]).
post_path(1,[p19]).
```

```
/* generating paths for unspecified receptions for example CTS */

/* loopless automatic generation to error_state */


error_path(1,[perror]).
error_path(2,[p1,perror]).
error_path(3,[p1,p6,perror]).
error_path(4,[p1,p7,perror]).
error_path(5,[p3,perror]).
error_path(6,[p3,p10,perror]).
error_path(7,[p4,perror]).
error_path(8,[p4,p10,perror]).


/* using 'fire_one' to walk through the specification */

walk(0,[p4,p10,p15,p15,perror]).
walk(1,[p1,p6,p13,p15,p14,perror]).
walk(2,[p3,p10,p13,p15,p14,p16,perror]).
```

APPENDIX C

TESTCASES

prologue

(1) tc_1_connection

There are 4 connection testcases. From the 'testcase_template' in the comment section of testcase 1 (i.e. file 'mar_20_931_pm') it is clear that 'connection(1,[p3,p12])' was obtained from a file of paths called 'transport_paths', that the output file was to be called 'mar_20_931_pm' and that the testcase value was to be 'pretty printed'. Note that the header extension_trace position is vacant in accordance with typicals.

```
Apr 18 20:18 1986  example_TCs/tc_1_connection/mar_20_931_pm Page
 1

path([p3,p12]).

path_value(0,[
[[[],[nsap,cr,[1,3171,[data],0,normal,64,980,128]]],[[tsap,tcind,
[178,980,64,9,normal,[data]]],[]]],
[[[tsap,tdreq,[474,reason]],[]],[[],[nsap,dr,[3171,1889,reason,1]
]]]
]).

          nsap
          cr
             1
             3171
                data

             0
             normal
             64
             980
             128

          tsap
          tcind
             178
             980
             64
             9
             normal
                data

          tsap
          tdreq
             474
             reason

          nsap
          dr
             3171
             1889
             reason
             1
```

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 5
4 : output_file    : mar_20_931_pm
5 : output_type    : pp
```

```
Apr 18 20:18 1986  example_TCs/tc_1_connection/mar_20_932_pm Page
 1

path([p4,p11]).

path_value(0,[
[[[],[nsap,cr,[1,322,[data],0,normal,378,260,0]]],[[tsap,tcind,[3
15,260,378,9,normal,[data]]],[]]],
[[[tsap,tcres,[315,10,normal,[]]],[]],[[tsap,tdind,[315,ts_qual_f
ail,[]]],[nsap,dr,[322,2591,[],3]]]]
]).

[[nsap,cr,[1,322,[data],0,normal,378,260,0]],[tsap,tcind,[315,260
,378,9,normal,[data]]]]
[[tsap,tcres,[315,10,normal,[]]],[tsap,tdind,[315,ts_qual_fail,[]
]],[nsap,dr,[322,2591,[],3]]]

comment
-------

the tc_template is:

1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 6
4 : output_file    : mar_20_932_pm
5 : output_type    : nl_write
```

```
Apr 18 20:18 1986. example_TCs/tc_1/connection/mar_20_933_pm Page
 1

path([p4,pl2]).

path_value(0,[
[[[],[nsap,cr,[1,647,[data],0,normal,33,202,0]]],[[tsap,tcind,[25
1,202,33,7,normal,[data]]],[]]],
[[[tsap,tdreq,[19,reason]],[]],[[],[nsap,dr,[647,3542,reason,1]]]]
]).

          nsap
          cr
             1
             647
                data

             0
             normal
             33
             202
             0

          tsap
          tcind
             251
             202
             33
             7
             normal
                data

          tsap
          tdreq
             19
             reason

          nsap
          dr
             647
             3542
             reason
             1

Comment
-------

the tc_template is:

1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 7
4 : output_file    : mar_20_933_pm
5 : output_type    : pp
```

```
Apr 18 20:18 1986 example_TCs/tc_1_connection/mar_20_934_pm Page
 1

path([p5]).

path_value(0,[[[[],[nsap,cr,[1,275,[data],0,expedited,600,345,0]]
],[[],[nsap,dr,[275,232,[data],3]]]]]).


          nsap
          cr
             1
             275
                data

             0
             expedited
             600
             345
             0

          nsap
          dr
             275
             232
                data

             3

Comment
-------

the tc_template is:

1 : input_file    : transport_paths
2 : input_functor : connection_establishment_path
3 : input_number  : 8
4 : output_file   : mar_20_934_pm
5 : output_type   : pp
```

(2) tc_2_data_transfer

Only one example for data transfer was made. 'fire_one' was used to generate the path which was filed in the internal database, this time, using the 'fout' function, under the name 'walking(0,[p3,p10,p13,p13,p14,p15,p16, p13,p14,p14,p18])'. Note that since 'fire_one' causes the present_state to change, the present-state is re-initialized with 'is.' before using 'mak_tc'. Also the input file for the testcase template is 'internal_db' which is a keyword that causes 'mak_tc' to search the internal database instead of an external file for the path.

```
Apr 18 20:19 1986  example_TCs/tc_2_data_transfer/mar_26_1039_pm
Page 1

path([p3,pl0,pl3,pl3,pl4,pl5,pl6,pl3,pl4,pl4,pl8]).

path_value(0,[
[[[],[nsap,cr,[1,322,[data],0,normal,378,260,128]]],[[tsap,tcind,
[315,260,378,9,normal,[data]]],[]]],
[[[tsap,tcres,[315,4,normal,[]]],[]],[[],[nsap,cc,[1,322,2591,[],
0,normal,378,260,0]]]],
[[[tsap,tdatr,[213,[207,231,158,27,226,231,174,188,83,25,200,156,
187,67,120,87,106,161,99,15,45,187,229,180],1]],[]],[[],[]]],
[[[tsap,tdatr,[355,[34,154,47,157,127,155,125,192,180,146,1,33,19
2,102,111,204,42,60,5,99,248,188,145,155,141,97,37,195,27,1,41,144
,30,135,217,154,240,27,187,214,19,64,17,232,66,50,4,238,195,236,31
,213,252,207,231,158,27,226,231,174,188,83,25,200,156,187,67,120,8
7,106,161,99,15,45,187,229,180,177,66,3,34,154,47,157,127,155,125,
192,180,146,1,33],0]]
,[]],[[],[]]],
[[[],[]],[[],[nsap,dt,[322,2591,[207,231,158,27,226,231,174,188,8
3,25,200,156,187,67,120,87,106,161,99,15,45,187,229,180],1,0]]]],
[[[],[nsap,dt,[2591,322,[data],1,1]]],[[],[]]],
[[[],[]],[[tsap,tdati,[281,[data],0]],[]]],
[[[tsap,tdatr,[360,[99,248,188,145,155,141,97,37,195,27,1,41],0]]
,[]],
[[],[]]],
[[[],[]],[[],[nsap,dt,[322,2591,[34,154,47,157,127,155,125,192,18
0,146,1,33,192,102,111,204,42,60,5,99,248,188,145,155,141,97,37,1
95,27,1,41,144,30,135,217,154,240,27,187,214,19,64,17,232,66,50,4
,238,195,236,31,213,252,207,231,158,27,226,231,174,188,83,25,200,
156,187,67,120,87,106,161,99,15,45,187,229,180,177,66,3,34,154,47
,157,127,155,125,192,180,146,1,33],2,0]]]],
[[[],[]],[[],[nsap,dt,[322,2591,[99,248,188,145,155,141,97,37,195
,27,1,41],3,1]]]],
[[[],[nsap,ndind,[1,nz_xdc]]],[[tsap,tdind,[315,ts_fail,[]]],[]]]
]).

        nsap
        cr
           1
           322
              data

           0
           normal
           378
           260
           128

        tsap
        tcind
           315
           260
           378
```

```
        9
        normal
          data

    tsap
    tcres
        315
        4
        normal
        []

    nsap
    cc
        1
        322
        2591
        []
        0
        normal
        378
        260
        0

    tsap
    tdatr
        213
            207
            231
            158
            27
            226
            231
            174
            188
            83
            25
            200
            156
            187
            67
            120
            87
            106
            161
            99
            15
            45
            187
            229
            180

        1

    tsap
```

```
tdatr
   355
      34
      154
      47
      157
      127
      155
      125
      192
      180
      146
      1
      33
      192
      102
      111
      204
      42
      60
      5
      99
      248
      188
      145
      155
      141
      97
      37
      195
      27
      1
      41
      144
      30
      135
      217
      154
      240
      27
      187
      214
      19
      64
      17
      232
      65
      50
      4
      238
      195
      236
      31
      213
```

252
207
231
158
27
226
231
174
188
83
25
200
156
187
67
120
87
106
161
99
15
45
187
229
180
177
66
3
34
154
47
157
127
155
125
192
180
146
1
33

0

nsap
dd

322
2591
207
231
158
27
226
231
174

```
                188
                83
                25
                200
                156
                187
                67
                120
                87
                106
                161
                99
                15
                45
                187
                229
                180

             1
             0

          nsap
          dt
             2591
             322
                data

             1
             1

          tsap
          tdati
             281
                data
             0

          tsap
          tdatr
             360,
                99
                248
                188
                145
                155
                141
                97
                37
                195
                27
                1
                41

             0
```

nsap dt
322
2591
34
154
47
157
127
155
125
192
180
146
1
33
192
102
111
204
42
60
5
99
248
188
145
155
141
97
37
195
27
1
41
144
30
135
217
154
240
27
187
214
19
64
17
232
66
50
4
238
195
236

```
31
213
252
207
231
158
27
226
231
174
188
83
25
200
156
187
67
120
87
106
161
99
15
45
187
229
180
177
66
3
34
154
47
157
127
155
125
192
180
146
1
33
2
0

nsap
dt
322
2591
99
248
188
145
155
```

```
                141
                97
                37
                195
                27
                1
                41

             3
             1

          nsap
          ndind
             1
             nz_xdc

          tsap
          tdind
             315
             ts_fail
             []
```

Comment
---

the tc_template is:

```
1 : input_file     : internal_db
2 : input_functor  : walk
3 : input_number   : 0
4 : output_file    : mar_26_1039_pm
5 : output_type    : pp
```

(3) tc_3_ur

For the general case, unspecified receptions may be treated as an error in the input interaction identifier. But there is a list of unspecified receptions available automatically at any state and status of the protocol by using ITSG's 'lur' command. With any of the 'lur' receptions, the tester may use the 'ur' command to generate an unspecified reception.

This directory has 4 example testcases (i.e. files). These were chosen from the unspecified reception list at the break in the input path [pl,break,perror].

The ITSG command 'lur' gives the following list at the state

```
[tsap,tcreq]
[nsap,cr]
[tsap,tcres]
[tsap,tdatr]
[nsap,dt]
[tsap,tdreq]
[nsap,ndind]
[nsap,nrind]
```

```
Apr 18 20:22 1986  example_TCs/tc_3_ur/mar_27_642_pm Page 1

path([pl,break,perror]).

extensions(ur([nsap,cr])).

extensions(input([nsap,cr,[1,2499,[data],0,normal,4,591,0]]),true
).

path_value(0,[
[[[tsap,tcreq,[232,322,378,0,normal,[]]],[]],[[],[nsap,cr,[1,3822
,[],0,normal,378,322,128]]]],
[[[nsap,cr,[1,2499,[data],0,normal,4,591,0]]],[[you are,in the,er
ror state]]]
]).
```

```
tsap
tcreq
   232
   322
   378
   0
   normal
   []

nsap
cr
   1
   3822
   []
   0
   normal
   378
   322
   128

nsap
cr
   1
   2499
      data

   0
   normal
   4
   591
   0

you are
in the
error state
```

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : error_path
3 : input_number   : 2
4 : output_file    : mar_27_642_pm
5 : output_type    : pp
```

```
Apr 18 20:22 1986  example_TCs/tc_3_ur/mar_27_647_pm Page 1

path([pl,break,perror]).

extensions(ur([tsap,tcres])).

extensions(input([tsap,tcres,[82,0,normal,'[]']]),true).

path_value(0,[
[[[tsap,tcreq,[213,44,487,1,normal,[]]],[]],[[],[nsap,cr,[1,414,[
],0,normal,487,44,128]]]],
[[[tsap,tcres,[82,0,normal,[]]]],[[you are,in the,error state]]]
]).

[[tsap,tcreq,[213,44,487,1,normal,[]]],[nsap,cr,[1,414,[],0,norma
l,487,44,128]]]
[[tsap,tcres,[82,0,normal,[]]],[you are,in the,error state]]

Comment
-------

the tc_template is:

1 : input_file    : transport_paths
2 : input_functor : error_path
3 : input_number  : 2
4 : output_file   : mar_27_647_pm
5 : output_type   : nl_write
```

```
Apr 18 20:22 1986  example_TCs/tc_3_ur/mar_27_649_pm Page 1

path([p1,break,perror]).

extensions(ur([tsap,tdatr])).

extensions(input([tsap,tdatr,[456,[137,67,120],1]]),true).

path_value(0,[
[[[tsap,tcreq,[498,966,28,7,normal,[]]],[]],[[],[nsap,cr,[1,3609,
[],0,normal,28,966,128]]]],
[[[tsap,tdatr,[456,[187,67,120],1]]],[[you are,in the,error state
]]]
]).

        tsap
        tcreq
           498
           966
           28
           7
           normal
           []

        nsap
        cr
           1
           3609
           [].
           0
           normal
           28
           966
           128

        tsap
        tdatr
           456
              187
              67
              120

           1

        you are
        in the
        error state
```

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : error_path
3 : input_number   : 2
4 : output_file    : mar_27_649_pm
5 : output_type    : pp
```

```
Apr 18 20:22 1986  example_TCs/tc_3_ur/mar_27_650_pm Page 1

path([pl,break,perror]).

extensions(ur([nsap,dt])).

extensions(input([nsap,dt,[3515,997,[data],_1116,0]]),true).

path_value(0,[
[[[tsap,tcreq,[128,953,403,5,normal,[]]],[]],[[],[nsap,cr,[1,2093
,[],0,normal,403,953,128]]]],
[[[nsap,dt,[3515,997,[data],_1114,0]]],[[you are,in the,error sta
te]]]
]).

        tsap
        tcreq
          128
          953
          403
          5
          normal
          []

        nsap
        cr
          1
          2093
          []
          0
          normal
          403
          953
          128

        nsap
        dt
          3515
          997
            data

          _1094
          0


        you are
        in the
        error state
```

Comment
----

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : error_path
3 : input_number   : 2
4 : output_file    : mar_27_650_pm
5 : output_type    : pp
```

(4) tc_4_atyp

This example uses the path [p1,p8] and we have selected to put anomalies (atyp) in the input interaction of the p8 transition. ITSG prompts where the tester would like to insert the break and we choose after p1. At the break, one may do various 'help/display' calls to decide on the extension atypical anomaly possibilities. We choose to put value-highs on the parameters 'dest_ref', 'source_ref', something unusual in 'user_data' and value-high in 'disconnect_reason'. This makes 4 testcases; 2 worked and 2 didn't. The resolution of the problems from the 2 that didn't are explained in the testers commehts in the actual testcases.

Apr 18 20:24 1986 example_TCs/tc_4_atyp/mar_29_1145_am Page 1

```
path([pl,break,p8]).

extensions(atyp).

extensions(dest_ref(65535),true).

path_value(0,[
[[[tsap,tcreq,[168,439,788,6,normal,[]]],[]],[{},[nsap,cr,[1,2364
,[],0,normal,788,439,128]]]],
[[[],[nsap,dr,[2364,517,[data],1]]],[[tsap,tdind,[168,ts_u_nrm,[d
ata]]],[nsap,ndreq,[1]]]]
]).

        tsap
        tcreq
            168
            439
            788
            6
            normal
            []

        nsap
        cr
            1
            2364
            []
            0
            normal
            788
            439
            128

        nsap
        dr
            2364        (* not what we wanted *)
            517
                data

            1

        tsap
        tdind
            168
            ts_u_nrm
                data

        nsap
        ndreq
            1
```

Comment
---

the tc_template is:

```
1 : input_file      : transport_paths
2 : input_functor   : connection_establishment_path
3 : input_number    : 7
4 : output_file     : mar_29_1145_am
5 : output_type     : pp
```

We wanted to insert an atypical value of high i.e. 65535, for parameter dest_ref in input dr_TPDU of p8 but it didn't occur. We checked p8 and found it was already instantiated from the additional variables of the present_state as LREF (local_reference) to the value 2364 and thus was not instantiated to 65535. This is an indication of the usefulness of ITSG to prevent a tester making involuntary errors. In this case, the cr tpdu was received from the transport protocol entity whose reference id is 2364. It is only logical to expect a dr tpdu from the same transport protocol entity and not from the entity with the reference id 65535.

Apr 18 20:24 1986 example_TCs/tc_4_atyp/mar_29_1201_pm Page 1

```
path([p1,break,p8]).

extensions(atyp).

extensions(source_ref(65535),true).

path_value(0,[
{[[tsap,tcreq,[165,64,980,1,normal,[]]],[]],[[],[nsap,cr,[1,1179,
[],0,normal,980,64,128]]]],
[[[],[nsap,dr,[1179,65535,[data],1]]],[[tsap,tdind,[165,ts_u_nrm,
[data]]],[nsap,ndreq,[1]]]]
]).

        tsap
        tcreq
           165
           64
           980
           1
           normal
           []

        nsap
        cr
           1
           1179
           []
           0
           normal
           980
           64
           128

        nsap
        dr
           1179
           65535          (* the atypical we wanted *)
              data

           1

        tsap
        tdind
           165
           ts_u_nrm
              data

        nsap
        ndreq
           1
```

Comment
----

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 7
4 : output_file    : mar_29_1201_pm
5 : output_type    : pp
```

```
Apr 18 20:24 1986  example_TCs/tc_4_atyp/mar_29_1221_pm Page 1

path([pl,break,p8]).

extensions(atyp).

extensions(user_data(['Hi, I''m Russ',is,this,list,short,or,'what
?']),
true).

path_value(0,[
[[[tsap,tcreq,[474,889,597,3,normal,[]]],[]],[[],[nsap,cr,[1,1563
,[],0,normal,597,889,128]]]],
[[[],[nsap,dr,[1563,2817,[Hi, I'm Russ,is,this,list,short,or,what
?],1]]],[[tsap,tdind,[474,ts_u_nrm,[Hi, I'm Russ,is,this,list,sho
rt,or,what?]]],[nsap,ndreq,[1]]]]
]).

          tsap
          tcreq
             474
             889
             597
             3
             normal
             []

          nsap
          cr
             1
             1563
             []
             0
             normal
             597
             889
             128

          nsap
          dr
             1563
             2817
                Hi, I'm Russ
                   is
                   this
                   list
                   short
                   or
                   what?

             1

          tsap
          tdind
```

```
            474
            ts_u_nrm
               Hi, I'm Russ
               is
               this
               list
               short
               or
               what?

         nsap
         ndreq
            1
```

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 7
4 : output_file    : mar_29_1221_pm
5 : output_type    : pp
```

```
Apr 18 20:24 1986  example_TCs/tc_4_atyp/mar_29_1225_pm Page 1

path([p1,break,p8]).

extensions(atyp).

extensions(disconnect_reason(5),true).

path_value(0,[
[[[tsap,tcreq,[374,680,54,7,normal,[]]],[]],[[],[nsap,cr,[1,3033,
[],0,normal,54,680,128]]]],
[[[],[nsap,dr,[3033,2202,[data],1]]],[[tsap,tdind,[374,ts_u_nrm,[
data]]],[nsap,ndreq,[1]]]]
]).

          tsap
          tcreq
             374
             680
             54
             7
             normal
             []

          nsap
          cr
             1
             3033
             []
             0
             normal
             54
             680
             128

          nsap
          dr
             3033
             2202
                data

             1              (* not what we wanted *)


          tsap
          tdind
             374
             ts_u_nrm
                data

          nsap
          ndreq
             1
```

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : connection_establishment_path
3 : input_number   : 7
4 : output_file    : mar_29_1225_pm
5 : output_type    : pp
```

Like atyp no. 1, previously, the value of the parameter is forced by the logic specification (see p8 to note that parameter disconnect_reason(1) is pre_instantiated and not allowed to vary even though the specification seems to allow values 1 through 5. After the comment we erase the anomaly from the database by calling it to avoid future unintended anomaly generation!

(5) tc_5_pe

In this case we used 3 examples. The first was a value error and the other 2 were format parameter errors. Format errors may be manufactured by using the 2-way function 'undo'. Given a clause tail 'undo' will turn the tail into a list and conversely, given a list, it will return a clause tail. Whatever list a tester makes can be turned into an executable tail of a clause which then can be inserted for execution and automatic deletion during execution by the function 'mouse'. So any format variation, larger or smaller (or otherwise than specified), can be created just by inventing a list. Similarly breaks can be inserted into a clause tail with the function 'add_breaks_clause' and so extension breaking is possible. Also extending the break by hopping to the actual extension location you want to change is easy to effect. Always a trace is kept and reported in the testcase extension trace section. The comment facility also becomes very handy when tester maneuvers and motivations get complicated! (see the testcases for comments and results).

```
Apr 18 20:24 1986  example_TCs/tc_5_pe/mar_29_126_pm Page 1

path([pl,break,perror]).

extensions(pe([nsap,dr])).

extensions(source_ref(65536),true).
extensions(input([nsap,dr,[1600,65536,[data],3]]),true).

path_value(0,[
[[[tsap,tcreq,[251,867,27,2,normal,[]]],[]],[[],[nsap,cr,[1,275,[
],0,normal,27,867,128]]]],
[[[nsap,dr,[1600,65536,[data],3]]],[[you are,in the,error state]]
]
]).

          tsap
          tcreq
             251
             867
             27
             2
             normal
             []

          nsap
          cr
             1
             275
             []
             0
             normal
             27
             867
             128

          nsap
          dr
             1600
             65536          (* here's the error *)
                data

             3

          you are
          in the
          error state
```

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : error_path
3 : input_number   : 2
4 : output_file    : mar_29_126_pm
5 : output_type    : pp
```

```
Apr 18 20:25 1986  example_TCs/tc_5_pe/mar_29_143_pm Page 1

path([pl,break,perror]).

extensions(pe([nsap,dr])).

extensions(input([nsap,dr,[_1112,_1113,_1114]]),(source_ref(_1112
)','u
ser_data(_1113)','disconnect_reason(_1114))).
extensions(input([nsap,dr,[2499,[data],2]]),true).

path_value(0,[
[[[tsap,tcreq,[232,322,378,0,normal,[]]],[]],[[],[nsap,cr,[1,3822
,[],0,normal,378,322,128]]]],
[[[nsap,dr,[2499,[data],2]]],[[you are,in the,error state]]]
]).

        tsap
        tcreq
            232
            322
            378
            0
            normal
            []

        nsap
        cr
            1
            3822
            []
            0
            normal
            378
            322
            128

        nsap        (* We have forced only 3 parameters where *)
        dr          (* there should be 4 *)
            2499
              data

            2

        you are
        in the
        error state
```

Comment
----

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : error_path
3 : input_number   : 2
4 : output_file    : mar_29_143_pm
5 : output_type    : pp
```

What we did at break was:

```
undo(A,[source_ref(B),user_data(C),disconnect_reason(D)]),
pe([nsap,dr],input([nsap,dr,[B,C,D]]),A).
```

lly and pe has 3 terms:

1/ [nsap,dr] - it is a specified reception so 'pe' doesn't detect an error

2/ and 3/ make up the substitutable clause - use this input instead of the input in the logic specification

```
Apr 18 20:25 1986  example_TCs/tc_5_pe/mar_29_222_pm Page 1

path([pl,break,perror]).

extensions(pe([nsap,dr])).

extensions(input([nsap,dr,[_1116,_1117]]),(dr_TPDU(_1116)','dt_TP
DU(_1117))).
extensions(input([nsap,dr,[[414,2587,[data],0],[999,1966,[data],_
1116,0]]]),true).

path_value(0,[
[[[tsap,tcreq,[86,213,44,7,normal,[]]],[]],[[],[nsap,cr,[1,1511,[
],0,normal,44,213,128]]]],
[[[nsap,dr,[[414,2587,[data],0],[999,1966,[data],_1114,0]]]],[[yo
u are,in the,error state]]]
]).

          tsap
          tcreq
             86
             213
             44
             7
             normal
             []

          nsap
          cr
             1
             1511
             []
             0
             normal
             44
             213
             128

          nsap
          dr
                414
                2587
                   data
                0

                999  (* nsap_dr has 2 parameters, here where the *)
                1966  (* first would succeed on its own but the *)
                   data (* 2nd causes failure                    *)

                _1094
                0

          you are
          in the
```

error state

Comment
---

the tc_template is:

```
1 : input_file     : transport_paths
2 : input_functor  : error_path
3 : input_number   : 2
4 : output_file    : mar_29_222_pm
5 : output_type    : pp
```

We did this as follows:

```
undo(A,[dr_TPDU(B),dt_TPDU(C)]),
pe([nsap,dr],input([nsap,dr,[B,C]),A).
```

APPENDIX D

ITSG LISTING

```
/* boot file */

go
   :-
     ['../itsg/help_procs',
      '../itsg/protocol_procs',
      '../itsg/utilities',
       '../lspec/tp_t/tp'],          /* transport lspec */
      general_startup,
      itsg.


/* initial_status file */

flow_ready.
i_am_ready.
```

```
/* help_procs file */

/* ----------- prompt ------------ */

itsg
  :-
    nl,nl,
    write('itsg: display system startup information by typing
          ''startup'''),
    repeat,
    telling(Z), tell(user),
    nl, write('itsg: '),
    (read(X)->                                        % I've got X
             (
             tell(Z),
             call(X) ->
                       true
                       ;
                       (
                          telling(Z),
                          tell(user),
                          nl,
                          write('no.'),
                          tell(Z)
                       )
              )
              ;
              true
    ),
    fail.

                          /*--------//--------*/

general_startup
  :- set(mode(i)).

startup
   :-
      nl,nl,
      write('startup steps'),nl,
      write('--------------'),nl,
      nl,
      write('(1) Consult your logic specification.'),nl,
      write('    eg. [''../lspec/tp_t/tp''].'),nl,
      nl,
      write('(2) Consult any other files you want.'),nl,
      write('    eg. [any_other_file, ''/u/grad/lucky''].'),nl,
      nl,
      write('(3) Display initial state with ''help(initial_state)''.
             It should be'),nl,
      write('    part of the logic specification with the form ''
             initial_state(LIST)''.'),nl,
      nl,
      write('(4) Display initial status with ''help(initial_status)'
```

```
            if it is '),nl,
        write('    part of the logic specification. It is of the form
              ''initial_status(LIST)''.'),nl,
        write('Otherwise'),nl,
        write('    enter ''status(F)'' to set the initial status
              interactively where F'),nl,
        write('    is the file containing all the status info.'),nl,
        nl,
        write('                    DISPLAY THIS MENU BY TYPING ''startup''
        nl,nl.


/* ---------- status -------------*/

status(File)  /* initial_status and present_status are control */
  :-          /* files */
   collect1(P,File),
   assert(initial_status(P)),
   assert(present_status(P)).

collect1([[X,Y]|I],File)
  :-
    see(File),
    read(X),
    (X==end_of_file -> fail
                    ; ( write(X), write(' , (on/off)? '),
                        see(user),read(Z), nl,
                       (Z==on -> (assert(X),Y=on)
                              ; Y=off),
                        collect1(I,File))).

collect1([],File) :- seen, see(user).


/* ------------ help procs ----------- */

show(X) :- help(X).

help
   :-
      nl,
      write('DISPLAY FUNCTIONS AVAILABLE'),
      nl,
      write('--------------------------'),
      nl, nl,
      write('1. help(display_functions).       or help(d). '),
      nl, nl,
      write('2. help(processing_functions).    or help(p). '),
      nl, nl,
      write('3. help(management_functions).    or help(m). '),
      nl, nl,
      write('NOTE: help may always be abbreviated to h'),
      nl.
```

```
h :- help.

h(X) :- help(X).

                        /*--------//--------*/

help(display_functions)
   :-
      nl,
      write('ITSG DISPLAY FUNCTIONS AVAILABLE'),
      nl,
      write('--------------------------------'),
      nl, nl,
      write('1. help. '), nl,
      write('2.  help(initial_status)   or help(iss). '), nl,
      write('3.  help(present_status)   or help(pss). '), nl,
      write('4.  help(initial_state)    or help(is). '), nl,
      write('5.  help(present_state)    or help(ps). '), nl,
      write('6.  help(mode).            or help(md).'), nl,
      write('7.  help(ID).  /* Display the transition or
                parameter */ '), nl,
      write('8.  lt.        /* List all transition ids     */ '), nl
      write('9.  lf.        /* List firables               */ '), nl
      write('10. lnf.       /* List non firables           */ '), nl
      write('11. lsr.       /* List specified receptions   */ '), nl
      write('12. lur.       /* List unspecified receptions */ '), nl
      nl,

      write('NOTE: help may always be abbreviated to h'),
      nl.

help(d) :- help(display_functions).

                        /*--------//--------*/

help(processing_functions)
   :-
      nl,
      write('PROCESSING FUNCTIONS AVAILABLE'),
      nl,
      write('-----------------------------'),
      nl, nl,
      write('1.  initialize_state.            or is.'), nl,
      write('2.  initialize_status.           or iss.'), nl,
      write('3.  set(present_state).          or set(ps).'), nl,
      write('4.  set(present_status).         or set(pss).'),
      nl,
      write('5.  set(mode(transition_id)).    or set(m(tid)).'),
      nl,
      write('6.  set(mode(interaction_id)).   or set(m(iid)).'),
      nl,
      write('7.  set(mode(interaction)).      or set(m(i)).'),
      nl,
      write('8.  set(tc_template).            or set(tct).'),
```

```
        nl,
        write('9.  auto_fire(Final_state, Output_type, Identifier).
        '),nl,
        write('    or  af(...same args...).'), nl,
        write('10. auto_fire(No_of_paths, Final_state,Output_type,
        Identifier).'),nl,
        write('    or  af(...same args...).'), nl,
        write('11. fire_all(Execution_path, Output_type,
                  Identifier).'),nl,
        write('    or  fa(...same args...).'),nl,
        write('12. fire_one(ID).                    or fo(ID).'), nl,
        write('13. fout(Output_type, Identifier).'), nl,
        write('14. backtrack(N)                     or bt(N).'),nl,nl,
        write('NOTE: set may always be abbreviated to s').

help(p) :- help(processing_functions).

                        /*--------//--------*/

help(management_functions)
   :-
        nl,
        write('MANAGEMENT FUNCTIONS AVAILABLE'),
        nl,
        write('------------------------------'),
        nl, nl,
        write('1.  set(trace_all)  / shut(trace_all).'), nl,
        write('2.  set(trace_fire) / shut(trace_fire).'), nl,
        write('3.  i_clean(In,Out)      /* cleans an interaction
                  list for output */'), nl,
        write('4.  add_break(N,List,Result_list). /* puts a break
                  in a path                  */'), nl,
        write('5.  add_break_clause(H,T,N,Res_T). /* puts a break
                  in a clause tail           */'), nl,
        write('6.  nl_write(A) / pp(A)            /* nl_write and
                  pretty_print               */'), nl,
        write('7.  top(Filename).'), nl,
        write('8.  shut(Filename).'), nl,
        write('9.  dump(Filename, Functor).'), nl,
        write('10. load(Filename, Head, Tail).'), nl,
        write('11. loadall(Filename, Head, Tail).'), nl,
        write('12. locate(Filename, Head, Tail).'), nl,
        write('13. listall(Filename, Functor).'), nl,
        write('14. listall(Filename).'), nl,
        write('15. atyp(Head, Tail).              /* atypical parameter
                  anomaly                    */'), nl,
        write('16. ur([AP, Iid]).                 /* unspecified
                  reception anomaly          */'), nl,
        write('17. pe([AP,Iid], Head, Tail).  /* parameter error
                  anomaly                    */ '), nl,
        write('18. mak_tc.                        /* make a test-case
                  */ '), nl.

help(m) :- help(management_functions).
```

```
                        /*--------//--------*/

% help procedures begin
/* 1 display(ID) ------- */

help(TID)                   /* TID is the transition id */
   :-
      clause(t(TID, PS, NS, I, O ), Y),
      write('Transition_Id: '),
      write(TID), nl,
      write('Present State: '),
      write(PS), nl,
      write('Next State: '),
      write(NS), nl,
      write('Input: '),
      write(I), nl,
      write('Output: '),
      write(O), nl,
      write('   :-'),nl,
      undo(Y),
      write('.'), nl.

/* 2 display_mode  ------- */

help(md) :- help(mode).

help(mode)
   :-
      mode(X),
      nl, write('mode is: '), write(X).

/* 3 display_system_state  -------- */

help(initial_state)
   :-
   initial_state(X),
   out(X).
help(is)
   :-
   help(initial_state).

help(present_state)
   :-
   present_state(X),
   out(X).
help(ps)
   :-
   help(present_state).

/* 4 display_system_status  ------- */

help(initial_status)
   :-
```

```
   initial_status(X),
   out(X).
help(iss)
   :-
   help(initial_status).

help(present_status)
   :-
   present_status(X),
   out(X).
help(pss)
   :-
   help(present_status).

/* lists  */
/* 5 list_all_transition_ids ---------*/

lt
   :-
      clause(t(TID, PS, NS, I, O), Y), nl,
      write(TID),write('    '), write(PS),
      fail.

lt.

/* 6 list_all_firables ----------------*/

lf
   :-
     present_state(PS),
      (mode(i) -> unhook ; true),
      t(TID,PS,_,_,_),nl,
      write(TID), tab(2), fail.

lf :- mode(i) -> hook ; true.

/* 7 list_all_non_firables ----------*/

lnf
   :-
    present_state(PS),
    (mode(i) -> unhook ; true),
    clause( t(TID, _,_, _, _ ), Y ),
    ( t(TID, PS, _, _, _ ) -> fail ; true ),
    nl, write(TID), tab(2), fail.

lnf :- mode(i) -> hook ; true.

/* 8 list_specified_receptions --------------*/

lsr
   :-
    present_state(PS),
    (mode(i) -> unhook ; true),
```

```
    clause(t(TID,PS,NS,I,_),Y),
    (var(I) -> error_stack([I|_]) ; true ),
    member([X1,X2,X3],I),
    nl, write(TID), tab(3), write([X1,X2,X3]),
    (t(TID,PS,NS,_,_) -> (nl, write('NS = '), write(NS))
                       ; (nl, write(' not firable in present
                                  status'))),
    nl, fail.

lsr :- mode(Y) -> hook ; true.

/* 9 list_unspecified_receptions ----------*/
/*
I first used the setof function:

'setof(I-I1, A^B^C^D^E^ clause(t(A,B,C,I1,D),E), X)'

is the equivalent 'setof' expression for the 'findall' expression
below except that 'findall' also includes empty cases!
'setof' was abandoned in favor of the more straight-forward
'findall'.
*/

lur :- lur(X),applist(out,X),nl.

lur(W1)
  :-
    findall(I1, clause(t(_,_,_,I1,_),Y1), X),  /* universe of
    receptions */
    ul(X,X1),      /* uniquize list          */
    present_state(PS),
    findall(I2, clause(t(_,PS,_,I2,_),Y2), Y),  /* specified
    receptions   */
    ul(Y,Y1),
    diff(X1,Y1,W),                 /* set difference      */
    de(W,W1).                      /* delete empty inputs. */

/*
X={ all possible input interactions }
Z={ all input interactions specified at PS }
W={ what we want with extra []'s }
*/

help(X) :- listing(X). /* last because 'listing' is a restriction
  of 'help' */
```

```
/* protocol processing procs file */

/*processing functions ---------*/

s(X) :- set(X).

/* initialize */

is :- initialize_state.

initialize_state
   :-
   initial_state(X),
   (retract(present_state(_));true), /* in case absent */
   assert(present_state(X)).

iss :- initialize_status.

initialize_status
   :-
   initial_status(X),
   retractall(present_status(_)),
   mimick(X),    /* present_status mimics initial status */
   assert(present_status(X)).   /* control template */

mimick([]).

mimick( [[H1,H2]|T] )
   :-
      retractall(H1),
      (H2=on -> assert(H1) ; true),
      mimick(T).

/* set system state */

/*
lists present_state as a numbered array and allows random access
updating by element number */

set(ps) :- set(present_state).

set(present_state)
   :-
      present_state(X),
      nl, write('the present_state is: '),nl,
      tab(3), numbered_output(1,X),nl,

      write('write q to quit'),nl,
      write('position: '),
      read(N),

(member(N,[q,quit]) -> true /* making case out of '->' */

(
```

```
integer(N),
!,          % a cut to prevent alternate successes
write('value for '),   % from a down-stream failure by
write(N), write(': '), % 'substitute'
read(Z),
(member(Z,[q,quit]) -> true

;

(
substitute(N,X,Z,Y),
retract(present_state(X)),
assert(present_state(Y)),
set(present_state)
)
)).

/*-----*/

set(pss) :- set(present_status).

set(present_status)
   :-
    retract(present_status(X)),
    !,                    /* only 1 choice however */
    collect2(X,Y),
    assert(present_status(Y)).

set(present_status) /*if no pss then set pss=iss and try again*/
   :-
    initialize_status(X), !, set(present_status).

collect2([],[]),

collect2([[H1,H2]|T], [[H1,NH2]|NT])
   :-
    write(H1),write(','),write(H2),write('     (y/n)?'),
    read(Z),
    (member(Z,[y,yes]) -> NH2=H2
                        ; (H2=on -> (retractall(H1),NH2=off)
                                  ;  (retractall(H1),assert(H1),
                                      NH2=on)
                           )),
    collect2(T,NT).

/*-----------------------------------------------------------------*/

/*
set(M) set_mode eg. TID, IID, or I_mode
*/

set(m(X)) :- set(mode(X)).

set(mode(X))
  :-
   (member(X,[transition_id,tid,interaction_id,iid,
```

```
                interaction,i ])
                -> true                          : /* recognize   */
                                 ; (nl,write('*** '), write(X),
                                    write(' is improper mode input'),
                                    fail)),
        (mode(Y) ; Y=[]),        /* what exists */
        mode_compare(X,Y).

/* mode_compare(X,X). */  /* its reset-able now! */

mode_compare(i,_) :- retractall(mode(_)), assert(mode(i)), hook.

mode_compare(X,_) :- retractall(mode(_)), assert(mode(X)), unhook.

/*-------------------------------------------------------------*/
/* auto_fire -----*/

/* counted exhaustive auto_fire */
/* auto_fire(Final_state, Output_type, Identifier) -----*/

af(FS,Output_type,Id) :- auto_fire(FS,Output_type,Id).

auto_fire(FS,Output_type,Id)
  :-
     reset(counter(do,1)),  /* afo uses do's counter */
     auto_fire_main(FS,Output_type,Id),
     inc(counter(do,1)),  /* keep count for printout */
     fail.

auto_fire(FS,Output_type,Id)
  :-
     auto_fire_last(FS, Output_type, Id).

/* do fail auto_fire */
/* auto_fire(No_of_paths, Final_state, Output_type, Identifier) */

af(N,FS,Output_type,Id) :- auto_fire(N,FS,Output_type,Id).

auto_fire(N,FS,Output_type,Id)
  :-
     do(N,auto_fire_main(FS,Output_type,Id)),
     auto_fire_last(FS, Output_type, Id).


/* main auto_fire */
/* auto_fire(Final_state, Output_type, Identifier) -----*/

auto_fire_main(FS, Output_type, Id)
  :-
    set(mode(tid)),
    present_state([H|T]), !,
/* we only want traversal repeatable */
    loopless_traversal([H|T],FS,[H],Path),
```

```
/* only keep major state list */
   auto_fire_output(Path,Output_type,Id).

/* loopless_traversal --------*/

/*  this traversal may be faster. after a fail at end we must be
able to restart back one step and not a continue as we originally
had, check_end checks to quit or recur ... and if quit followed
later by a fail it backtracks one and restarts
*/

loopless_traversal(PS, FS, Already_visited, [TID|TID_tail])
   :-
    t(TID,PS,[H|T],I,O),
    not(member(H,Already_visited)),
    append([H], Already_visited, New_Already_visited),
    check_end([H|T], FS,New_Already_visited, TID_tail).

/* NS=[H|T] so H is a major state variable */

check_end([H|T], FS,New_Already_visited, TID_tail)
   :-
    (H=FS -> (TID_tail=[])
          ; loopless_traversal([H|T], FS, New_Already_visited,
            TID_tail)).

/*--auto_fire main output---*/

auto_fire_output([TID|TID_tail], Output_type, Identifier)
   :-
     counter(do,N),
     afo(N,[TID|TID_tail],Output_type,Identifier), !.

afo(N,P,1,Id) :- A=..[Id,N,P], out(A).          /* screen */

afo(N,P,OT,Id) :- (OT=2 ; OT=3), A=..[Id,N,P], assert(A).

/* auto_fire tail output */
/* just checks if output_type 3; so 'dump' called */

auto_fire_last(FS, 1, Identifier).

auto_fire_last(FS, 2, Identifier).

auto_fire_last(FS, 3, Identifier)
   :- write('give me a filename: '),
      read(F),
      dump(F,Identifier).

/*-----------------------------------------------------------------*/

/* fire_all(Execution_path, Output_type, Identifier) */

fa(Path,Output_type,ID) :- fire_all(Path,Output_type,ID).
```

```
fire_all(Path,Output_type,ID)
  :-
    present_state(PS),
    set(mode(i)),
    fire_list(Path,PS,Interactions),
    fire_all_output(Interactions,Output_type,ID).

fire_list([],_,[]).

fire_list([H|T],PS,[[I,O]|Y])
  :-
    (H=TID ; H=[I,O]),
    t(TID,PS,NS,I,O),
    fire_list(T,NS,Y).

fire_list([break|T],PS,[[I,O]|Y]) /* we can have a break in the
        path now */
   :-
     retract(present_state(M)),
     assert(present_state(PS)),
     break,
     retract(present_state(_)),
     assert(present_state(M)),
     fire_list(T,PS,[[I,O]|Y]).

fire_all_output(Interactions,1,ID) :- A=..[ID,0,Interactions],
                                      out(A).

fire_all_output(Interactions,2,ID) :- A=..[ID,0,Interactions],
                                      assert(A).

fire_all_output(Interactions,3,ID)
   :-
     A=..[ID,0,Interactions],
     write('give me a filename: '),
     read(F),
     tell(F),
     out(A),
     told.

/* fire */
/*
fire_one(ID) automatically sets reverse_path trace
*/

fo(ID) :- fire_one(ID).

fire_one(ID)
   :-
    (mode(i) -> (nl, write('set the mode to tid or iid'),fail) ;
                true), present_state(PS),
    ( ( ID=TID, t(TID,PS,NS,I,O) )
      ;
      ( ID=[IPI,II], t(TID,PS,NS,I,O), member([IPI,II,IPL],I) )
```

```
    ),
    retract(present_state(PS)),
    assert(present_state(NS)),
    (retract(reverse_path(Y)) ; Y=[]), /* in case 1st time     */
(mode(tid) -> assert(reverse_path([TID|Y]))/* according to mode*/
              ;  assert(reverse_path([[I,O]|Y]))).

/* ---------------*/

fout(OT,ID)     /* special dump for fire_one*/
  :-
    retract(reverse_path(X)),
    reverse(X,Y),
    A=..[ID,0,Y],
    foutout(OT,A).

foutout(1,A) :- out(A).

foutout(2,A) :- assert(A).

foutout(3,A)
   :-
     write('give me a filename: '),
     read(F),
     tell(F),
     out(A),
     told.

/* backtrack(N) --------*/

bt(N) :- backtrack(N).

backtrack(N)
  :-
    present_state(NS),
    reverse_path([H|T]),
    backup(N,NS,[H|T]).

backup(N,NS,[H|T])
  :-
    N>0,
    (H=TID ; H=[I,O]),
    t(TID,PS,NS,I,O),
    M is N-1,
    (M>0 -> backup(M,PS,T)
        ; (retract(present_state(_)),
           assert(present_state(PS)),
           retract(reverse_path(_)),
           assert(reverse_path(T)))
    ).

/* error transition ---------------------- */

t(perror, [H|T], [error_state|NS_tail], [I], [['you are','in the'
```

```
,'error state']])
  :-
    not(H=final),
    input(I).

/* atypical and erroneous extension filling processes */

atyp(A, B)    /* an atypical - reception is specified */
  :-
     path(X),
     not(member(perror,X)),          /* atypical = error! */
     assert(extensions(atyp)),       /* trace             */
     mouse(A,B).

ur([A,B])
  :-
     path(X),     /* 2 tests to see if 'ur' is allowed  */
     member(perror,X),   /* 1/ path is indeed erroneous */
     lur(Y),      /* 2/ chosen reception is unspecified */
     member([A,B,_],Y),
     assert(extensions(ur([A,B]))),  /* trace */
     input([A,B,C]),  /* make the input now and mouse it    */
     mouse(input([A,B,C]),true).
/* parameters of the head replaced only */

/*pe(specified reception, substitution rule for protocol error)*/
/*pe([AP,I], head, [tail_list] )                               */

pe([M, N], A, B)    /* parameter error */
  :-
     path(X),                                   /* 2 tests */
     member(perror,X),

     present_state(PS),
     (mode(i) -> unhook ; true),
     findall(II,t(_,PS,_,II,_),Y),
     (mode(i) -> hook ; true),
     member_member([M,N,_],Y),

     assert(extensions(pe([M,N]))),       /* trace   */
     mouse(A,B),
     input([M,N,C]),
     mouse(input([M,N,C]),true).

member_member(A,[H|T])
   :-
      not(free_var(H)), member(A,H) ; member_member(A,T).

free_var(X) :- X=[A], var(A).  /* perror always has wild IID */

/* mouse asserts a rule to the database which if executed
retracts itself - good for momentary distortions. Might
put in a counter etc. in the future. */
```

```
/* mouse(Head,Tail) is a like clause(Head,Tail)

I have a handy little function - 'undo(B,[H|T])' - which packs or
unpacks.  Its a 2-way function:
     eg.      ','(a,','(b,c)) <---> [a,b,c].
     i.e. undo(B, B_list)
So if we have a list we can pack it to 1 element (as prolog does)
and visa-versa we can undo a clause tail to a usable list.
*/

mouse(A,B)
  :-
    assert(extensions(A,B)),                    /* trace */
    asserta(     (A:-B,retract( (A:-B,retract(_)) )   )   ).

/*-----------------------------------------------------------------*/
/* set test case template - very similar to 'set(ps)' */

/*
lists tc_template as a numbered array and allows random access
updating by element number
*/

set(tct) :- set(tc_template).

set(tc_template)
  :-
     Y=['input_file   ', 'input_functor', 'input_number ',
        'output_file  ', 'output_type  '],
     (tc_template(X) ->true
                     ; (X=[_, _, _, _, _],assert(tc_template(X))
                       )),
     nl, write('the tc_template is: '),nl,
     tab(3), numbered_output(1,Y,X),nl,

    write('write q to quit'),nl,
    write('position: '),
    read(N),

(member(N,[q,quit]) -> true     /* making case out of '->' */
                     ;
(
integer(N),
!,                      % a cut to prevent alternate successes
write('value for '),    % from a down-stream failure by
write(N), write(': '),  % 'substitute'
read(Z),
(member(Z,[q,quit]) -> true
                     ;
(
substitute(N,X,Z,Q),
retract(tc_template(X)),
assert(tc_template(Q)),
set(tc_template)
```

```
))
)).

/*-----*/

mak_tc
  :-
    set(tc_template), !,
    tc_template([INFILE, FUNCTOR, NO, OUTFILE, OUTTYPE]),
    nonvar(INFILE),        /* they must be real values */
    nonvar(FUNCTOR),
    nonvar(NO),
    nonvar(OUTFILE),
    nonvar(OUTTYPE),
    retractall(path(_)),
    retractall(path_value(0,_)),
    retractall(extensions(_)),
    retractall(extensions(_,_)),
    load_path(INFILE, FUNCTOR, NO),
    path(Z),
    fire_all(Z,2,path_value),!, /* don't enter fire_all again */
    (member(perror,Z) -> check_ur_or_pe ; true),
    path_value(0,X),
    i_clean(X,Y),
    tell(OUTFILE),
    /* comment-maker goes here */
    nl,nl,nl, h(path),
    nl,nl,nl, h(extensions),
    nl,nl,nl,
/*  h(path_value),  */
( path_value(N,L) -> (nl, write('path_value('),
                      write(N), write(','),
                      nl_write_list(L), write(').')
                     )
                   ; true),
    nl,nl,nl,
    (OUTTYPE = nl_write -> applist(nl_write,Y) ; pp(Y,0)),
    told.

check_ur_or_pe :- extensions(ur(_)).

check_ur_or_pe :- extensions(pe(_)).

check_ur_or_pe :-
  nl_write('err** path indicates error but none was generated'),
  fail.

load_path(File, Functor, Number)
  :-
    A=..[Functor, Number, X],
    (File=internal_db -> A          /* for external gets */
                       ; ( top(File), locate(File,A,_))),
    (member(perror,X) -> add_break(X,Z)
                       ; (nl, write(X), nl,
```

```
                write('insert break after what position? '),
                      write('      or -1 for no break '),
                      read(N),
                      (N>=0 -> add_break(N,X,Z) ; Z=X))
        ),
        B=..[path,Z],
        assert(B).
```

```
/* management procs and utilities file */

/* itsg specific functions */
/*-----------------------------------------------------------------*/

   trace_all i.e. auto_fire trace
   trace_fire i.e. fire_one trace
   trace_extension or trace_I/O (not implemented) ------------*/

/* update_trace(list_identifier, element_to_be_added_to_list) */

update_trace(A,B)
  :-
    trace_add(A,->,B) ; (trace_add(A,<-,B),fail).

/* trace_add(list_identifier, direction,
   element_to_be_added_to_list) */

trace_add(X,Y,Z)
  :-
    B=..[X,M],
    (retract(B) ; M=[]),    /* pick-up or make it [] */
    N=..[X,[[Y,Z]|M]],      /* pack list with functor */
    assert(N), !.

/*-----------------------------------------------------------------*/
/* dump(Filename, Functor)-- save is system word!----------
dumps all database elements (rules or facts) with that functor
to the file and deleting the internal database of those elements
*/

dump(Filename, Functor)
  :-
    tell(Filename),
    current_functor(Functor,S),
    clause(S,Y),
    retract((S:-Y)),
    out((S:-Y)),
    fail.

dump(_,_) :- told.

nl_write(A) :- nl,write(A).

out((X:-Y)) :- nl, (Y=true -> write(X);write((X:-Y))), write('.')
,!.

out([A]) :- nl, write([A]), write('.').  /* lists == clauses
                                            quite! */

out(X) :- out((X:-true)).

/* load(Filename, Element)-------------*/
```

```
load(Fname, X, Y)    /* if (X:-Y) not appropriate use load(F,X,Y) */
   :-
     top(Fname),
     find(Fname,X,Y),
     seen,
     (Y=end_of_file -> (nl,write('***not found - searched whole
                                  file'),nl)
                     ;   assert((X:-Y))).


loadall(Fname, X,Y)  /* user instantiates X, Y */ /* rules */
   :-                          /* as he wishes -          */
     top(Fname),
     repeat,
     clause_file(Fname,X,Y),
     (Y=end_of_file -> true ;  (assert((X:-Y)),fail)  ).

/*-----------------------------------------------------------------*/
/* hook and unhook affect I/O and hooking tells which is
   currently set */

unhook :- hook, asserta((input(_) :- !)),
asserta((output(_) :- !)).

hook :- retractall((input(_) :- !)),retractall((output(_) :- !)).

hooking(X) :- clause(input(_),!),   /* for user information */
              clause(output(_),!),
              X = unhook.

hooking(X) :- X = hook.

/* itsg independent */
/*-----------------------------------------------------------------*/

top(File) :- seeing(X), see(File), seen, see(X).

shut(File) :- tell(File),told.

type(File)
  :-
     see(File),
     repeat,
     get0(X),
     (X=26 -> ( !,seen)
          ; (put(X), fail)
     ).

type(File,Line_count)
   :-
     retractingall(line_count(_)),
     integer(Line_count), Line_count > 0,
     assert(line_count(Line_count)),
     seeing(X),
```

```
        see(File),
        iteration,
        see(X).

iteration
    :-
        repeat,
        line_count(N),
        get0(X),
        (N>0 ->(
                X=26 -> true
                      ; (
                        ( X=10 -> (
                                   retract(line_count(N)),
                                   M is N-1,
                                   assert(line_count(M))
                                   )
                                 ;
                                   true
                        ),
                        put(X),
                        fail
                        )
                )
                ;
                true
        ),!.


/* locate ------------ */

locate(F, H, B) :- clause_file(F, H, B).

clause_file(File,Head,Body)     % Body=end_of_file if not found
    :-                          % Body=true if a fact
      seeing(X),                % Body is the wanted answer
      find(File,Head,Body),
      see(X).

find(File,Head,Body) % given a Head it finds a corresponding Body
    :-               % or, at end_of_file, Body is assigned end_of_file
      see(File),
      repeat,
      read(Term),
      (Term=end_of_file -> Term=Body
                         ;
                           (Term =(Head:-Body) -> true
                                                ; (Term=Head,Body=true)
                           )
      ),!.                 % cut for one-way, like I/O is!

/* listall---------- */
```

```
listall(Fl, F2) :- typing(Fl, F2).

typing(File,Functor)
  :-
     repeat,
     find(File,Head,Body),
     (var(Head), Head=..[Functor|_]),   /* at eof head is var */
     (Body=end_of_file -> true
                        ;(
                          write(Head),
                          (Body=true ->(write('.'),nl)
                                       ;( write(Head),nl,
                                          write(' :-'),nl,
                                          undo(Body),write('.'),nl,
nl
                                        )),
                          fail
                         )
     ).

listall(Fl) :- typing(Fl, _).

ls :- system("ls").

/*-------------- gem_procs --------------*/

%used in values_db

mak_list(_, 0, []).
mak_list(X, 1, X).
mak_list(X, N, Str) :- N>1, M is N-1, mak_list(X, M, Str2),
                       append(X, Str2, Str).

concat([X|[]], X).
concat([X|Y], Z) :- append(X, W, Z), concat(Y, W).

append([], X, X).
append([X|X1], X2, [X|X3]) :- append(X1, X2, X3).

member( X, [X|_] ).
member( X, [_|Y] ) :- member( X, Y).

retractall(X) :- retract(X), fail.

retractall(X) :- retract((X:-Y)), fail.

retractall(_).

/* pretty printing -2- for lists!
X is list, I is column ptr, tab(3) was used as interval spacer
*/

pp(X,I) :- var(X), !, tab(I), write(X),nl.
```

```
pp([H|T],I) :- !, J is I+3, pp(H,J), ppx(T,J),nl,!.

pp(X,I) :- tab(I), write(X),nl.

ppx([],_).

ppx([H|T],I) :- pp(H,I), ppx(T,I).

/*
given a predicate-name and a list, 'assert_list' asserts the list
element by element to the database with the predicate name given.
*/

assert_list(_, []).

assert_list(Predicate,[L|Ltail])
    :-
      append([Predicate], L, L1),
      Z=..L1, /* make list a predicate for prolog */
      assert(Z),
      assert_list(Predicate,Ltail).

/*
write_list
*/

nl_write_list([]) :- write([]).

nl_write_list([H|[]]) :- write([H]).

nl_write_list([H|T])
      :-
        write('['),
        nl_write(H),
        write(','),
        nl_write_comma(T).

nl_write_comma([H|[]]) :- nl_write(H), nl, write(']').

nl_write_comma([H|T])
    :-
      nl_write(H), write(','), nl_write_comma(T).

/* retracts the above assert_list */

retract_list(_, []).

retract_list(Functor,[L|Ltail]) /* for a predicate over a list of
 arguments */
    :-
      Z=..[Functor|L], /* make list a predicate for prolog */
      retract(Z),
      retract_list(Functor,Ltail).
```

```
same_len(X,M) /* is list X same length as M? */
   :- list_len(X,N), N==M.

/* length of a list */

list_len([],0).

list_len([X|[]],1).

list_len([X|Y],N) :- list_len(Y,M), N is M+1.


undo(X)              % undo the body of a clause for listing print
out.
   :-
      ( X=..[A|[B|C]] ->
                     (A=',' ->
                              (write('      '), write(B), write(','),
                               nl, C=[D], undo(D)
                              )
                              ;
                              ( X=(D), write('      '), write(D) )
                     )
                     ;
                     (X=(D), write('      '), write(D)
                     )
      ).

/* undo a tail i.e. make it into a list */

undo(Y,[H|T])                    /* 3 elements */
   :-
      Y=..[',',H,B],
      undo(B,T),!.

undo(Y,[Y|[]]) :- !. /* 1 element  */
/* it also catches-all but here occurs only after
a ','  operator of the clause tail fails    */


undo_write([H|T]) :- nl, tab(5), write(H).

/* map i.e. function a list */

maplist(_,[],[]).

maplist(P,[X|L],[Y|M]) :- Q=..[P,X,Y], call(Q), maplist(P,L,M).

/* apply a list */

applist(_,[]).

applist(P,[X|L]) :- Q=..[P,X], call(Q), applist(P,L).
```

```
/* difference a list */

diff([],B,[]).

diff([A|AT],B,C) :- member(A,B), !, diff(AT,B,C).

diff([A|AT],B,[A|CT]) :- diff(AT,B,CT).

/* reverse a list */

reverse([],[]).

reverse([H|T],L) :- reverse(T,Z), !, append(Z,[H],L).

/* uniquize a list */

ul([],[]).

ul([A|B],L) :- member(A,B),!, ul(B,L).

ul([A|B],[A|L]) :- not(member(A,B)), !, ul(B,L).

/* delete_empty */

delete_empty([],[]).

delete_empty([H|T],Z) :- H=[], delete_empty(T,Z).

delete_empty([H|T],[H|Z]) :- not(H=[]), delete_empty(T,Z).

/* decrement little counters called
'counter(name,No. of iterations)'
to start 'assert(counter(Counter_id,No:))' and 'dec'
fails after N 'decs' */

dec(counter(Counter_id,N))
  :-
    retract(counter(Counter_id,M)),
    N is M-1,
    N >=0,
    assert(counter(Counter_id,N)).

/* interactions can be cleaned of empties and their transition
character by 'maplist(de,IN,OUT)' which applies 'de' to each
I/O of an interaction path */

i_clean(IN,OUT) :- maplist(de,IN,OUT)

/* Hasan's delete_empty for ITSG *
de([],[]).

de([H|T],R)
  :- de(T,P),
     findall([X,Y,Z],member([X,Y,Z],H),L),
```

```
    append(L,P,R).

/*----------------------*/

/* 'substitute' and 'number_output' are used in system_state
     substitute(position number,
                given list,
                replacement,
                resulting list) */

substitute(_,[],_,_) :- fail.

substitute(1,[H|T], R, [R|T]).

substitute(N,[H|T], R, [H|L]) :- N > 1, M is N-1,
                                 substitute(M,T,R,L).

/* makes an array of the list with its index value explicit */

numbered_output(_,[]).

numbered_output(N,[H|T])
  :-
    nl, write(N), write(' : '), write(H),
    M is N+1,
    numbered_output(M,T).


/* 3 args */
numbered_output(_,_,[]).

numbered_output(N,[X|Y],[H|T])
  :-
    nl, write(N), write(' : '), write(X), write(' : '), write(H),
    M is N+1,
    numbered_output(M,Y,T).

/*-------------------------*/
/* do_loops  - to be used to avoid big recursion expressions when
 we wish */

doo(N,A)       /* for one-sided predicates like 'write'    */
  :-           /* we use 'repeat' to force it to try again */
  M is N+1,
  reset(counter(doo,0)),
  repeat, A,
  inc(counter(doo,1)),  /* incs before test so M =  N +
                           counter_step  */
  counter(doo,M).

do(N,A)
  :-
  M is N+1,
  reset(counter(do,1)),
```

```
  A,
  inc(counter(do,1)),   /* incs before test so M = N +
                           counter_step */
  counter(do,M).

reset(counter(ID,N))
  :-
     retractall(counter(ID,_)),
     assert(counter(ID,N)),!.

inc(counter(ID,N))
    :-
     retract(counter(ID,M)),
     J is M+N,
     assert(counter(ID,J)),!.

/*-------------------------*/

/* idea is:        { X | predicate } = L
       i.e.    findall(X,  Predicate,    L)
like in set theory!
== built-in predicate 'setof(X,Y,Z)'
*/

findall(X,G,_)
  :-
   asserta(found(mark)),
   call(G),
   asserta(found(X)),
   fail.
findall(_,_,L) :- collect_found([],M),!,L=M.

collect_found(S,L) :-getnext(X),!,collect_found([X|S],L).
collect_found(L,L).

getnext(X) :- retract(found(X)), !, X==mark.


/* add_break to list after N elements of the list */
/* add_break(after_position, List, New_list)      */

ab(X,Y) :- add_break(X,Y).


/* automatic position */
add_break([perror|T],[break,perror|T]).

add_break([H|T],[H|Q]) :- add_break(T,Q).

/* with position */
add_break(0,X,[break|X]).

add_break(N,[H|X],[H|Y]) :- M is N-1, add_break(M,X,Y).
```

```
/* add_break_clause(Head,Tail_original,Number,Tail_new) */
/* unpacks and packs a tail of a clause with the purpose of
inserting a 'break' in it */

abc(H,T1,N,T2) :- add_break_clause(H,T1,N,T2).

add_break_clause(H,T1,N,T4)
  :-
    N>=0,
    clause(H,T1),
    undo(T1,T2),
    add_break(N,T2,T3),
    undo(T4,T3).

/* retract first break in a list */
/* retract_break(List, New_list)  */

rb(X,Y) :- retract_break(X,Y).

retract_break([],[]).

retract_break([break|Y],Y).

retract_break([H|X],[H|Y]) :- retract_break(X,Y).
```