Method of analysing extended finite-state machine specifications

Behçet Sarikaya, Vassilios Koukoulidis and Gregor V Bochmann* develop
a technique for self-consistency analysis of complex protocol/service specifications

Formal specifications are the basis for automated verification and implementation in communication software. The paper gives a method of dynamic analysis for modular specifications which is based on symbolic execution and reachability analysis. Symbolic execution is a technique for static analysis and applied first to the specification. It is effective in detecting syntactic and semantic errors. A form of reachability analysis, called limited reachability, is used to dynamically analyse the intermodule communication. It has two applications: combining modules and detecting any errors in intermodule communication. The technique is first applied to the specifications in a nondeterministic finite-state machine model and then applied to an extended finite-state machine model for which two standard formal description languages exist.

Keywords: specification, implementation, communication software, symbolic execution, limited reachability, finite-state machine model

Formal specification of communication systems is considered to be of prime importance in both software and hardware development. This is because formal specifications are not ambiguous and they describe the system precisely, as opposed to natural language specifications which often suffer from being ambiguous and imprecise. It is also possible to base the automated system verification and implementation on formal specifications.

An area in which formal specifications are finding widespread use is communication protocols for Open Systems Interconnection (OSI). Formal techniques are being used for the specification of the proposed protocols, and the development and testing of the new protocol implementations. Three FDTs are presently being used: Estelle[12], Lotos[15], and SDL[23]. Estelle is based on a finite-state machine model extended by Pascal data structures, expressions and statements for the description of interaction parameters, additional state variables and related processing. Lotos is based on a calculus of communicating systems extended with a formalism for abstract data types. SDL is also based on an extended finite-state machine model. At present, SDL is more widely used due to its graphical syntax called SDL-GR.

Formal specifications of standardized protocols and services are being developed to be used as reference or as specifications complementary to the traditional reference specifications given in natural language. Since all subsequent phases depend on the formal specification, validation of formal specifications is of prime importance.

Validation of specifications can be classified into two activities: validation of self-consistency and validation of consistency with another specification. In both cases, the validation could be based on static analysis or dynamic analysis. Static analysis looks for syntactic and semantic errors in the specification without considering an execution of the specification. Dynamic analysis looks for system deadlocks, unspecified receptions and other problems arising from the dynamic behaviour of the specified system[5].

Concordia University, Department of Electrical and Computer Engineering, 1455 De Maisonneuve W. #915, Montréal, Québec, Canada H3G 1M8
*Université de Montréal, Dépt d'IRO, CP 6128, Succ. 'A', Montréal, Québec, Canada H3C 3J7

0140-3664/90/020083-10 $03.00 © 1990 Butterworth & Co (Publishers) Ltd

In this paper a method is described for self-consistency validation of specifications written in an extended finite-state machine model. The basic techniques used are symbolic execution, a well-known method used in analysis of sequential programs[10] and reachability analysis[24, 13], a well-known technique used in protocol verification for protocols specified as finite-state machines. Two techniques are then used to dynamically analyse modular specifications.

The paper is organized as follows: we first introduce the limited reachability analysis, a reachability analysis with the consideration of the queues of length one. An algorithm is given for limited reachability analysis of modular specifications in which modules are specified in a nondeterministic finite-state machine model. The symbolic execution of formal specifications in an extended FSM model is then discussed, and then extended to the limited reachability analysis to this model.

REACHABILITY ANALYSIS FOR COMMUNICATING FSMs

Reachability analysis

Reachability analysis is a technique often used to analyse the dynamic behaviour of multiprocess systems defined as a collection of interacting finite state machines (FSMs). Traditionally, models of directly interacting FSMs[3, 17] as well as models involving FSMs communicating through (unlimited) FIFO queues[18, 24] have been considered. In order to partly reduce the state space explosion involved with a straightforward exploration of all attainable global states of the system, the concept of 'reduced' reachability has been proposed[18, 25]. In general, there remains the possibility of unlimited build-up of messages in the queues, which makes the general validation problem undecidable. In many restricted cases, however, many validation problems can be decided even in the case of unlimited queue lengths[14].

The application of this technique for protocol validation usually implies the analysis of a system, as shown in Figure 1a, of two interacting protocol entities (PEs). Only the protocol data units (PDUs) exchanged between the two entities are considered. For a complete protocol validation, based on a protocol specification also involving the interactions with the service users, a configuration as shown in Figure 1b must be considered. In addition, it is shown in this paper that the specification of a single protocol entity is partitioned into several communicating FSMs. This corresponds to typical specifications written in FSM-oriented FDTs, such as Estelle or SDL, which typically contain several (sometimes dynamically created) interacting processes (see, for instance, Reference 4).

Reachability analysis for embedded systems

Consider the case of a system consisting of a large number of communicating FSMs; interest lies in validating the interactions between FSMs belonging to a certain subsystem, independently of the other parts of the system. An example is the validation of the specification of a protocol entity given as a collection of interacting FSMs, as shown in Figure 2, or two adjacent protocol entities within a given host computer. If reachability analysis is considered for such an embedded (sub)system, independently of the other system components, one will not be able to detect errors which relate to the interactions between the subsystem and the other parts of the system, e.g. the same information is not obtained as in the case of traditional protocol reachability analysis which considers the interactions with the peer entity; however, one will still be able to validate those aspects of the specification which relate to the interactions among the FSMs within the same subsystem.

The traditional work on reachability analysis uses a FSM formalism with what are called 'simple' transitions, that is,

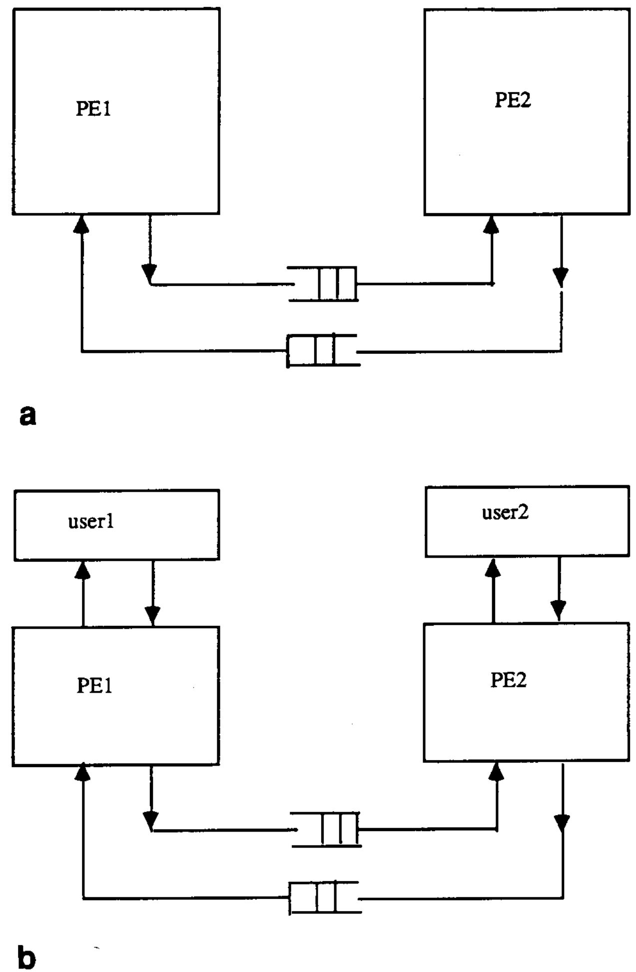


Figure 1. Architectures for protocol validation

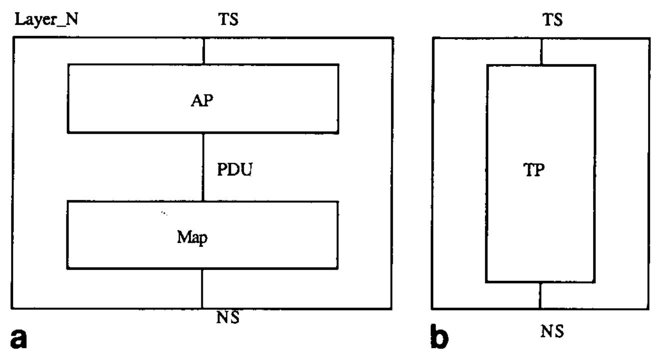


Figure 2. Modular system structure (a) and its combined representation (b)

each transition is either an input transition, i.e. a transition for which an input is defined and no output is specified, or a spontaneous transition, i.e. a transition for which no input is defined and an output is specified. In the case of directly interacting FSM (i.e. rendezvous), the reachability analysis reduces to the construction of a coupled product of the interacting machines, as explained in Reference 17. The result of the reachability analysis is called in the following simply the 'product machine'. In the case of direct interaction, the product machine is again finite. In the case of FSMs interacting through queues, the product machine is not necessarily finite, unless the size of the queues is artificially limited.

The initially obtained product machine contains in general a certain number of transitions which involve the input/output between the interacting machines. These interactions are not visible from the environment of the analysed subsystem and can be considered spontaneous transitions without output to the environment. A straightforward simplification of the product machine can be performed in order to obtain an equivalent machine without such spontaneous transitions.

Limited reachability analysis

In the following a more general FSM model is considered, corresponding to the FDT's Estelle and SDL, where a single transition (for input or spontaneous) may involve one or more outputs. These FDTs use queued communication between the FSMs.

A reachability analysis is considered which assumes direct interaction between the machines, which is called 'limited reachability analysis'. An investigation is carried out in the following into what extent limited reachability analysis can be considered complete, that is, what kind of errors are not detected by such analysis and could be detected by reachability analysis taking into account message queueing.

Figure 2a depicts an example specification containing two modules: *AP* connected to the environment with the channel TS, and Map connected to the environment with the channel NS. In Figure 2b the product machine, called TP, is shown with the internal interaction point eliminated. AP and Map are interconnected with the channel PDU. Map FSM is shown in Figure 3 and AP FSM in Figure 4 with the state 'closed' represented twice to increase readability. This system models a transport protocol (TP) organized in two modules: an abstract protocol (AP) module handling protocol operations and a mapping (Map) module to map the transport protocol interactions to the network and vice versa.

In what follows, the notation for input/output:

channel.interaction(parameters) or
  interaction(parameters if any)

is used for internal channels (PDU in Figure 2) and external channels (TS and NS in Figure 2), respectively. Interaction is the name of the input/output which may have one or more parameters. Note that a textual syntax can be used to express the transitions in the FSMs using four different clauses: when, from, to and output. As an example, the transition in Figure 3 from the 'closed' state can be written as follows:

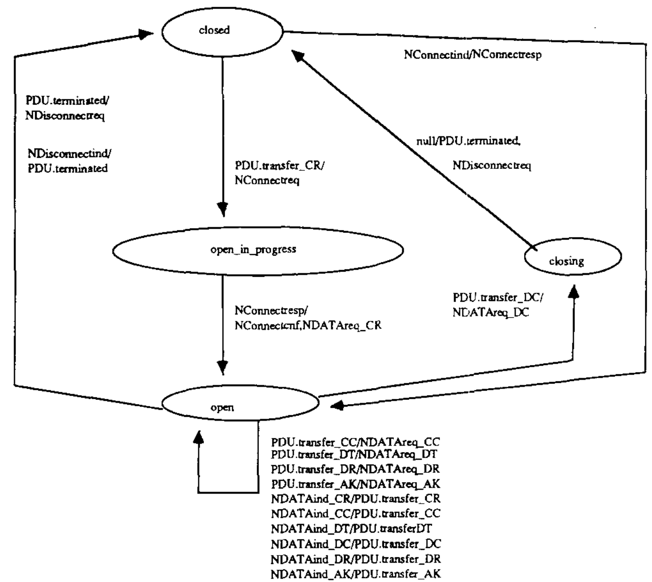


Figure 3. Map machine

```
when PDU.transfer_CR
from closed to open_in_progress
  output NS. NConnectreq.
```

Firstly, a simple example is given and then an algorithm for limited reachability sketched. Consider the following transition from Figure 4:

```
when TS.TCONreq
from closed to wait_for_CC
  output PDU.transfer_CR
```

This transition when fired (upon the input TCONreq from the channel TS) places the interaction called transfer_CR into the queue. A corresponding transition from Map is immediately considered that consumes this input:

```
when PDU.transfer_CR
from closed to open_in_progress
  output NS.NConnectreq.
```

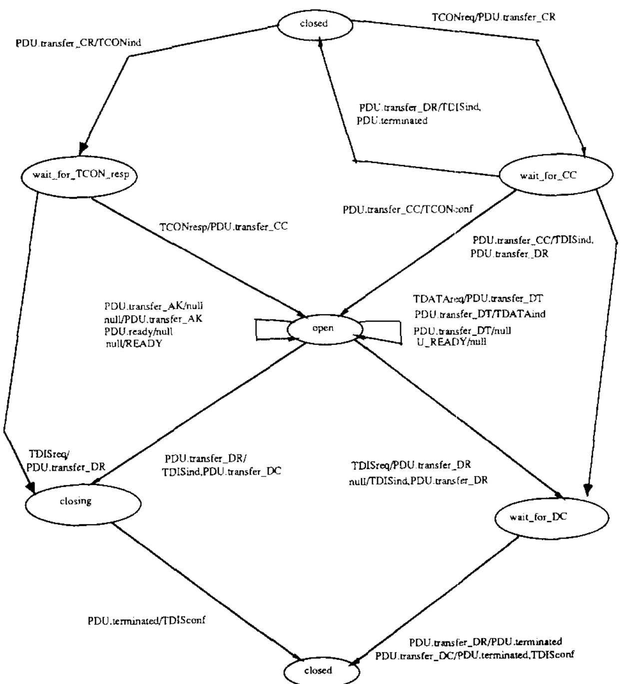


Figure 4. AP machine

Thus, the combined behaviour of the two modules can be expressed as a transition:

```
when TS.TCONreq
from (closed,closed) to (wait_for_CC,
  open_in_progress)
   output TS.NConnectreq
```

which corresponds to the case that after AP places transfer_CR into the queue, transfer_CR is consumed by Map, which in turn produces an output to the NS channel.

Algorithm for limited reachability

An algorithm is presented to analyse modular specifications with each module specified as a FSM. The algorithm contains three steps of processing.

First, all transitions which take input from external channels or spontaneous transitions are considered. If they produce an output to the internal channel, then these transitions are combined with the transitions in the other module which consume this output. This constitutes the first part of the Step 1 of the algorithm.

Next, the transitions combined in Step 1 are considered. The combined transitions that generate output to the internal channels need to be processed. These transitions are combined with the corresponding transitions that consume the output, completely eliminating the communication in the internal channel. This constitutes the second part of the Step 1 and corresponds to the case of double handshaking between the modules.

In Step 2, the transitions that neither take input nor produce output to the internal channels are considered next. These transitions generate more than one transition in the final product machine, i.e. one for each state value of the other FSM.

Since not all the state pairs occur in the product machine, the algorithm generates a combined transition for all reachable state pairs. Then null transitions (input and output are null) could also be eliminated. These actions constitute Step 3 of the algorithm listed in the Appendix. The algorithm is repeatedly applied to other channels interconnecting the FSMs (if any).

Error cases are not considered in the above algorithm, but they could easily be added so that the algorithm generates a list of erroneous cases in the combined behaviour. The types of errors that can be detected by the algorithm are:

- unspecified receptions, when there exists no transition for an output,
- handshaking loops, when for example, FSM 1 generates, upon input A the output B to the internal channel and FSM2 generates, upon input B the output A to the internal channel,
  deadlocks, when the product machine goes to a state with no outgoing transitions and the global state is not composed of the final states of the individual FSMs (if any).

For systems with more than two modules, it is straightforward to obtain the product machine by first combining any two interconnected modules and then combining the product machine with a third interconnected module, and so on.

Implementation

The above algorithm has been implemented on a workstation in Prolog. The program takes an Estelle specification of the embedded system in which the interconnected modules can be described as FSMs as input and produces another Estelle specification with modules combined. Description of FSMs in Estelle form is straightforward[6]. The user provides the names of the input file and the names of the modules to be combined. The program finds the channel name over which these modules communicate from the IP (interaction point) definitions of the modules and then follows the algorithm to generate combined transitions.

Example

It is shown how the implementation above analyses the combined behaviour of by way of the FSMs of Figures 3 and 4. The input Estelle specification looks like:

```
specification example systemprocess;
channel PDU_c (user, provider);
  by user: terminated; transfer_AK; . . . ; ready;
  by provider: terminated; transfer_AK; . . . ;
  transfer_DT;
channel TS(user, provider);
  by user: TCONreq; . . . ; U_READY;
  by provider: TCONind; . . . ; READY;
channel NS(user, provider);
  by user: NCONNECTind; . . . ; NDATAind_DT;
  by provider: NCONNECTreq; . . . ; NDATAreq_DT;
. . . (*module header definitions for the englobing
     module and its two submodules*)
. . . (*rename the channel PDU_c as PDU in the ip
     definitions*)
. . . (*body definition for the FSM of Figure 3*)
. . . (*body definition for the FSM of Figure 4*)
modvar AP:A; MAP:M;
initialize
  begin
    init AP with AA; init MAP with MM;
    connect AP.PDU to MAP.PDU
  end(*of initialize*)
end; (*of body*)
end. (*of specification*)
```

An example processing in Step 1 is combining

```
when NS.NDATAind_DR
from open to open
  output PDU.transfer_DR
```

in Figure 3 with

```
when PDU.transfer_DR
from open to closing
  output TS.TDISind
  output PDU.transfer_DC
```

In Figure 4 to obtain:

```
when NS.NDATAind_DR
from open_open to closing_open
  output TS.TDISind
  output PDU.transfer_DC
```

where the *from* or *to* lists contain first a state from the AP and next another state from the Map machines with the

names separated with '__' to have syntactically correct Estelle state names.

The combined transition is furthermore processed in Step 1 to be combined with the transition in Figure 3:

```
when PDU.transfer__DC
from open to closing
   output NS.NDATAreq__DC
```

resulting in the combined transition:

```
when NS.NDATAind__DR
from open__open to closing__closing
   output TS.TDISind
   output NS.NDATAreq__DC.
```

One of the transitions processed in Step 2 is:

```
when NS.NConnectind
from closed to open
   output NS.NConnectresp
```

of Figure 3. This transition generates six transitions corresponding to six states of the AP module. In Step 3 only the transition:

```
when NS.NConnectind
from closed__closed to closed__open
   output NS.NConnectresp
```

is kept since *closed__closed* is the only possible state pair.

Since there are no errors detected we represent the product machine in graphical form in Figure 5.

Comparison with full reachability analysis

Limited reachability analysis can be seen as a reachability analysis technique in which transitions taking input from the internal channel(s) have higher priority of consideration. Since the queues are eliminated from the global state, it is not possible to consider parallel progress of the component modules with limited reachability analysis. A consequence of this is that the resulting combined machine represents a part of the overall behaviour of the system.

An example case will be presented in which the combined machine does not represent the complete behaviour. For example, in Figure 1, the AP can receive an input from TS and the Map can receive another input from NS at the same time while this situation is not permitted in Figure 2. It seems that the only behaviour that is missing in the combined machine of Figure 5 is the call collision case, i.e., a connection establishment request arriving at the same time at both service boundaries.

Reduced reachability analysis can also handle collision cases since queues of equal length are considered as part of the global state. So far, it can only be applied to two-process communication while limited reachability analysis can consider any number of machines.

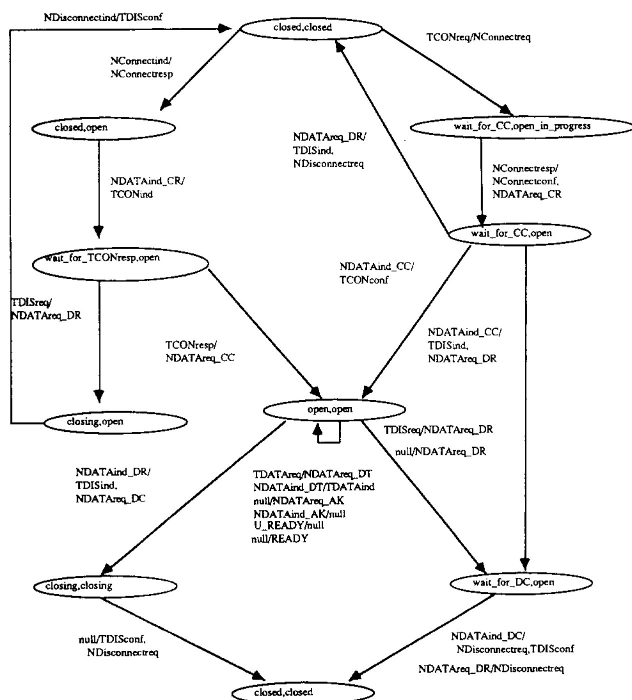


Figure 5. Combined FSM TP

Use of limited reachability analysis for embedded systems

Embedded systems can be validated using limited reachability analysis. Each subsystem is assumed to be modelled with FSMs which makes our analysis technique applicable. In the next section, it will be shown that a similar analysis is possible even when the specification technique is based on an extended finite-state machine model.

Limited reachability analysis first looks for any errors in the embedded system and then obtains a product machine. The product machine can be used in test sequence generation since only external channels are of interest for testers. The product machine could also be used in further verification of layered systems.

Since limited reachability analysis cannot handle collision cases, the product machine does not fully represent the embedded system. Therefore, collision cases should be handled separately when the product machine is used, i.e. test sequence generation should consider making a test for call collision.

LIMITED REACHABILITY FOR EFSM

In this section, the limited reachability analysis is extended to apply to the specifications written in an extended finite-state machine model (EFSM). This model first introduced in Reference 2 specifies a system using finite-state machines to represent major state changes with each transition of the system and context variables to represent actions attached to each transition. There exist two languages which are based on EFSM: Specification and Description Language (SDL) of CCITT[23] and Estelle of ISO[12].

In general, the specifications in the EFSM model may contain:

- major state lists or sets in FROM clauses,
- local procedures and functions which can be called anywhere from the transitions (PROVIDED clauses or the actions),
- statements changing the control of execution such as IF and CASE or loop statements (such as WHILE and

FOR) which can be included in the actions and/or in local procedures/functions,

- statements for dynamic module creation/release e.g., connect/disconnect and attach/detach clauses in Estelle,
- multiple modules with channels interconnecting them for inter-module communication. These channels are double queues in Estelle and single queues in SDL.

Symbolic execution

Symbolic execution is a technique in which a program is executed over symbols rather than actual values[10]. It has been used to statically analyse sequential programs as well as in automated test data selection. The result is the output in symbolic form along with path predicates of the path that generates this output.

Symbolic execution can also be used to dynamically analyse the program behaviour for the verification purposes[9]. It has been used for protocol verification in Reference 7, in which a proof tree is built by symbolically executing various processes specified as sequential programs. The proof is completed with respect to the assertions added to various points in the programs.

In order to apply the results of the previous section to the modular specifications in the EFSM model, it is desirable to have the specification in a form that clearly identifies the execution paths that generate outputs. It will be shown that it is possible to transform (using a form of static symbolic execution) a given specification into a normal form where transitions contain single paths. Such a specification is called a normal form specification containing normal form transitions (NFT)[19, 22].

The process of obtaining a normal form specification is performed by a set of basic transformations which are explained below. The example specification used is the simplified Class 2 transport specification in Reference 4. This specification written in Estelle is structured in two modules called AP (Abstract Protocol) and Map (Mapping). FSM representation of the AP module is given in Figure 4. The FSM representation of the Map machine has a single state (open), but as far as symbolic execution is concerned it is more complex than the AP module due to the use of several *for* and *while* loops. This specification will be referred to in the following as TP2 specification.

Basic transformations can be automated[1] and may detect various syntactic and semantic errors in the specification since syntactic and semantic checks must be done to ensure a correct specification. The resulting normal form specification facilitates test sequence generation[21]. It will be shown later in this section that the same technique can be used to analyse modular specifications.

Basic transformations

Major state lists/sets in FROM clauses such as:

```
[AKWAIT, OPEN, OPEN_WFEA]
```

are used to specify multiple initial states for transitions in the EFSM. Therefore these lists/sets can easily be eliminated by generating one NFT corresponding to each possible state value (state values of AKWAIT, OPEN and OPEN_WFEA in the above example).

IF and CASE statements are removed by generating an NFT for each path they define. Symbolic execution should be applied to the assignment statements in cases where conditional statements occur in places other than the first statement and their condition is on variables assigned before the conditional statements in the same transition. Such a case occurs for example in the transition (extracted from the TP2 specification):

```
WHEN Map.transfer
FROM open TO same
PROVIDED PDU.kind = AK
   var new_credit:pos_integer;
BEGIN
   new_credit:= credit_value
   + expected_send_sequence — TSseq;
   if new_credit >= S_credit
   then S_credit:= new_credit
   else (*error*)
END;
```

which, when symbolically executed transforms into the following two normal form transitions:

```
WHEN Map.transfer
from open to same
PROVIDED PDU.kind = AK and (credit_value
+ expected_send_sequence — TSseq >= S_credit)
1:BEGIN
   S_credit:= credit_value
   + expected_send_sequence — TSseq;
END;
WHEN Map.transfer
from open to same
PROVIDED PDU.kind = AK and (credit_value
+ expected_send_sequence — TSseq < S_credit)
2:BEGIN
(*error*)
END;
```

The loop statements are eliminated by repeating the body of the loop for every index variable value. As an example, the FOR statement:

```
for kind := CR to AK do PDU_buffer[kind].
is_last_PDU:=false;
```

generates, with the enumeration of all possible values for the variable Kind:

```
PDU_buffer[CR]:is_last_PDU:=false;
PDU_buffer[CC].is_last_PDU:=false;
        .
        .
PDU_buffer[AK].is_last_PDU:=false;
```

In cases where exhaustive enumeration is not possible, a limited number (usually three) executions of the loop body is considered. For example, the statement:

```
ref := 1;
while ref in active_refs do ref := ref + 1;
```

where active_refs is of type set of integers, could be transformed to:

```
ref := 1; for active_refs = φ
ref := 2; for active_refs = {1}
ref := 3; for active_refs = {1, 2}.
```

Local procedure/function calls are eliminated by

symbolically executing the local procedure/function body. Local variables of the procedures/functions are made global. More details on the treatment of parameters in procedure/function call elimination can be found in Reference 1.

The basic transformations described have been implemented on a workstation. The resulting system takes an Estelle specification as input, does lexical and semantic analysis on the specification. Only the correct specifications are subjected to the transformations. The transformations part of the system has been implemented in Prolog[16].

Analysing modular specifications in EFSM

Dynamic behaviour of different modules of a specification in EFSM can be analysed using limited reachability analysis explained above. The analysis looks for possible problems such as deadlocks, unspecified receptions, blocking receptions, tempo-blockings, etc. It is assumed that the specification is transformed into a normal form. The same steps are applied to the NFTs as in the algorithm given in the Appendix to find the combined transitions.

Modified algorithm

In Step 1, the spontaneous NFTs and the NFTs that consume input from external channels are considered. If any of these NFTs produce an output to the internal channel (called *combiner* NFT) it is combined with the NFT which consumes this output (called *combinee* NFT). Combining two NFTs is done as follows.

First the combinee NFT is processed by symbolic replacements for parameter values from the output statement of the combiner NFT in the PROVIDED clause and possibly in the action. Then the combined NFT is formed from the modified combinee NFT and the combiner NFT.

FROM and TO clauses are processed as in the algorithm of the Appendix. The PROVIDED clause of the combinee NFT is added in conjunction to the PROVIDED clause of the combiner NFT. The action of the combinee NFT replaces the output statement in the action part of the combiner NFT.

The processing in Steps 2 and 3 is essentially the same as in the algorithm of the Appendix with NFT combination as explained above. Since the combined transitions access to the context variables of the individual modules, these variables are made global to the combined module.

This modified algorithm has been implemented on a workstation. The program takes a transformed specification containing two interconnecting modules and obtains an output specification with these modules combined. It is also written in Prolog[16]. The resulting system is being used as part of a test generation system based a methodology which takes single module specifications as input[20].

Example

As an example, the TP2 specification is considered, and it is shown how the modified algorithm applies to it. Consider the following transition of the AP module:

```
when TS.TCONreq
from closed to wait_for_CC
begin
   options:= proposed_options;
   output Map.transfer(CR_PDU(to_T_address,
   options, R_credit));
end;
```

where CR_PDU is a local function defined as:

```
function CR_PDU(to_adr:TAddrType, o:OptType,
c:SeqNumType):TPDUandCtrlInf;
var PDU: TPDUandCtrlInf;
begin with PDU do
   begin kind := CR;
      peer_address := to_adr;
      options_ind := o;
      credit_value := c;
      order := first
   end;
CR_PDU := PDU;
end;
```

After the CR_PDU is symbolically replaced:

```
when TS.TCONreq
from closed to wait_for_CC
begin
   options := proposed_options;
   PDU.kind := CR;
   PDU.peer_address := to_T_address;
   PDU.options_ind := options;
   PDU.credit_value := R_credit;
   PDU.order := first;
   output map.transfer(PDU);
end;
```

The output statement is removed in Step 1 by combining it with the transition of the Map module:

```
when AP.transfer
begin
   TC[T_suf,EP_id]. PDU_buffer[PDU.CR] := PDU;
   TC[T_suf,EP_id]. PDU_buffer[PDU.CR]
   .full := true;
end;
```

yielding:

```
when TS.TCONreq
from (closed, idle) to (wait_for_CC, idle)
begin
   options := proposed_options;
   PDU.kind := CR;
   . . . (*same as above*)
   TC[T_suf,EP_id]. PDU_buffer[PDU.CR] := PDU;
   TC[T_suf,EP_id]. PDU_buffer[PDU.CR]
   .full := true;
end;
```

After the modules are combined, control and data flow graphs of the resulting specification can be obtained if there are no errors detected. For our example protocol specification, the FSM model of the product EFSM (after errors are corrected, see below) is similar to Figure 4, since the map module has a single major state, therefore this graph is not shown to save space.

Errors resulting from intermodule communication

Limited reachability analysis gives rise to detecting various problems in intermodule communication, thereby in the design of entity (such as the protocol/service) specifications. These problems can be classified as deadlocks, unspecified receptions and channel overflows. These problems are analysed by giving examples from an earlier version of the TP2 specification.

Deadlock arises when the channel is empty and either of the modules is unable to progress due to major and context state values, i.e. none of the transitions can be fired. In TP2 such a deadlock is detected when the AP module goes to 'closing' state after receiving a TDISreq from TS channel in 'wait__for__TCONresp' state and transfers a DR PDU to the 'map' module (see Figure 4) and the 'is__last__PDU' parameter is mistakenly set to false. The 'map' module in turn sends a NDATAreq to the NS channel instead of a 'terminated' output to the internal channel (since 'is__last__PDU' is set to false). The result is a deadlock since the AP module will indefinitely wait for 'terminated' signal to arrive in the internal channel.

Unspecified reception occurs when one of the modules places an input to the channel for which the other channel has no internal transition to consume. This case can also occur for external channels, i.e. an input primitive arriving in an unexpected state is usually left unspecified. On the other hand, there are transitions that must exist in the specification. If they are somehow forgotten, the result is unspecified receptions. In TP2, limited reachability detected unspecified reception errors in two cases: The AP module in 'wait__for__CC' state goes to 'wait__for__DC' upon reception of TDISreq at the TS channel and produces a DR PDU for 'map' module (see Figure 4) and sets (mistakenly) the 'is__last__PDU' parameter to true. This makes the 'map' module output a NDATAreq to the NS channel and a 'terminated' to the internal channel. The 'terminated' signal in 'wait__for__DC' state is an unspecified reception for the AP module. The second case occurs when the AP module goes to 'wait__for__DC' state from 'open' state with TDISreq input at TS channel.

In case one of the modules can repeatedly fire a transition which outputs an interaction to an external or internal channel, there is channel overflow. Channel overflows usually result from spontaneous transitions that generate an output. These transitions usually reflect an abstraction level in which the specifier is trying to capture various possible implementation behaviours. An example is the acknowledgement policy. The formal specification accepts all possible policies by letting the protocol entity to output an acknowledgement any time. A channel overflow related to external channels occurs in the TP2 specification where the AP module is allowed to send READY message on the TS channel indefinitely. Another channel overflow related to internal channels exists where the AP module sends AK PDU to the internal channel indefinitely. In limited reachability analysis this transition is combined with the corresponding reception transition in the 'map' module which in turn enables a transition that outputs a NDATAreq to the NS channel. These interactions produce, in the product machine, a spontaneous transition that outputs a NDATAreq to the NS channel. Thus the limited reachability analysis converts the channel overflows related to internal channels channel overflows related to the external channels.

The above discussion abstracts out the parallel behaviour description in Estelle by way of *systemprocess* and *systemactivity* properties of the modules. This aspect of Estelle is discussed in References 8 and 11.

CONCLUSIONS

A method was developed for self-consistency analysis of complex protocol/service specifications. Static validation based on symbolic execution reveals syntactic and some semantic errors while dynamic analysis based on reachability analysis reveals problems in inter-module communication such as deadlocks, channel overflows, unspecified receptions, etc. while the translator of the specification language could only detect static errors. The specification is first transformed into a form called normal form specification by the basic transformations of symbolic execution. Then modules are combined by a limited reachability analysis in order to eliminate internal communication and obtain a single module specification. The resulting single module specification can be used for test sequence generation as well as for further validations of complicated system specifications.

There is a need for developing a tool that will assist the specification developers by automatically doing most of the analysis described in this paper. This need becomes more evident in coping with volumunious specifications. Parts of the automatic processing can also be incorporated in the translators for the specification language. It would also be interesting to investigate if a similar technique would apply for self-consistency analysis of Lotos specifications.

The paper assumes the existence of queues of internal interactions that may contain a maximum of one message. Further theoretical investigations are required to extend our technique to the case where internal queues may have an arbitrarily large number of messages.

ACKNOWLEDGEMENTS

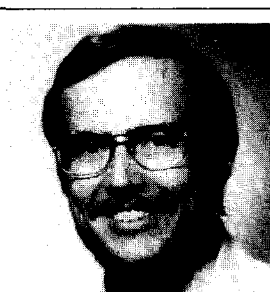
The authors are grateful to Rick To of McGill University for implementing the limited reachability algorithm. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1 **Barbeau, M and Sarikaya, B** 'A computer-aided design tool for protocol test design' *Proc. INFOCOM '88* (March 1988) pp 1D.3.1-10
2 **Bochmann, G v and Gecsei, J** 'A unified method for the specification and verification of protocols' *Proc. IFIP 77* (1977) pp 229-234
3 **Bochmann, G v** 'Finite state description of communication protocols' *Comput. Netw.* Vol 2 (1978) pp 361-372
4 **Bochmann, G v** 'Specifications of a simplified transport protocol using different formal description techniques' *Comput. Netw. ISDN Syst.*
5 **Bochmann, G v** 'Usage of protocol development tools: the results of a survey' *Proc. 7th IFIP Symp. on Protocols* North-Holland, The Netherlands (May 1987) pp 139-161

6 **Bochmann, G v and Vaucher, J** 'Adding performance aspects to specification languages' *Proc. 8th IFIP Symp. on Protocols* North-Holland, The Netherlands (June 1988) pp 19-31
7 **Brand, D and Joyner, W H** 'Verification of protocols using symbolic execution' *Comput. Netw.* Vol 2 (1978) pp 351-360
8 **Budkowski, S and Dembinsky, P** 'An introduction to Estelle: a specification language for distributed systems' *Comput. Netw. ISDN Syst.* Vol 14 No 1 (1987) pp 3-23
9 **Cheatham, T E, Holloway, G H and Townley, J A** 'Symbolic evaluation and the analysis of programs' *IEEE Trans. Softw. Eng.* Vol SE-5 No 4 (July 1979) pp 402-417
10 **Clarke, L A and Richardson, D J** 'Symbolic evaluation methods for program analysis' in **Muchnick, S S and Jones, N D (Eds)** *Program Flow Analysis* Prentice Hall, USA (1981)
11 **Courtiat, J P** 'Estelle*: a powerful dialect of Estelle for OSI protocol description' *Proc. 8th Symp. on Protocols* North-Holland, The Netherlands (1988) pp 171-186
12 'Estelle: A FDT based on an extended state transition model' ISO TC97/SC16/WG1, IS 9074 (November 1988)
13 **Gouda, M and Yu, Y T** 'Maximal progress state exploration' *Proc. SIGCOMM '83* Austin, Texas (March 1983) pp 68-73
14 **Finkel, A and Rosier, L** 'A survey on FIFO nets' Technical Report Univ. de Montréal (October 1987)
15 'Lotos: A FDT based on the temporal ordering of observational behaviour' ISO TC 97/SC 16/WG1 IS 8807 (1988)
16 **Koukoulidis, V** 'Full implementation of a test design methodology for protocol testing' MSc Thesis Concordia Univ. (March 1989)
17 **Merlin, P and Bochmann, G V** 'On the construction of submodule specification and communication protocols' *ACM TOPLAS* Vol 5 No 1 (January 1983) pp 1-25
18 **Rubin, J and West, C H** 'An improved protocol validation technique' *Comput. Netw.* Vol 6 (1982) pp 65-73
19 **Sarikaya, B** 'Test design for computer network protocols' PhD Thesis, McGill University (March 1984)
20 **Sarikaya, B, Barbeau, M, Eswara, S and Koukoulidis, V** 'A formal description based test generation tool' *Technical Report. Concordia University* (July 1988)
21 **Sarikaya, B, Bochmann, G v and Cerny, E** 'A test design methodology for protocol testing' *IEEE Trans. Softw. Eng.* (May 1987) pp 710-721
22 **Sarikaya, B and Bochmann, G v** 'Obtaining normal form specifications for protocols' *Proc. COMNET '85* Budapest, Hungary (October 1985) pp 601-612
23 'Specification and description language (SDL) — CCITT Recommendation Z100 Geneva, Switzerland (October 1987)
24 **Zafiropoulo, P, West, C H, Rudin, H and Cowan, D D** 'Towards analyzing and synthesizing protocols' *IEEE Trans. Comm.* Vol COM-28 No 4 (April 1980) pp 651-661
25 **Zhao, J R and Bochmann, G v** 'Reduced reachability analysis of communication protocols: a new approach' *Proc. 6th IFIP Workshop on Protocols* North-Holland, The Netherlands (June 1986) pp 243-254

***Behçet Sarikaya** received a BSEE degree from the Middle East Technical University (METU), Ankara, Turkey in 1973, a MS degree in Computer Science from METU in 1976, and a PhD degree in Computer Science from McGill University, Montréal, Canada in 1984. He was an assistant professor at the Université de Sherbrooke, Sherbrooke, Canada. He is presently an Assistant Professor in the Department of Electrical and Computer Engineering, Concordia University, Montréal, Canada. His current research interests lie in communication protocol specification and testing, and formal specification of communication systems. Dr Sarikaya is a member of the IEEE Computer Society and the Association for Computing Machinery.*



***Gregor v Bochmann** received a Diploma degree in Physics from the University of Munich, Munich, West Germany, in 1968 and a PhD degree from McGill University, Montreal, PQ, Canada, in 1971. He has worked in the areas of programming languages, compiler design, communication protocols, and software engineering and has published many papers in these areas. He is currently a Professor in the Département d'Informatique et de Recherche Opérationelle, Université de Montréal, Montréal. His present work is aimed at design models for communication protocols and distributed systems. He has been actively involved in the standardization of formal description techniques for OSI. From 1977-78 he was a Visiting Professor at the Ecole Polytechnique Federale, Lausanne, Switzerland, followed by a similar position in the Computer Systems Laboratory, Stanford University from 1979-80. The period 1986-87 saw Professor Bochmann take up the post of Visiting Researcher at Siemens, Munich.*

APPENDIX: AN ALGORITHM FOR LIMITED REACHABILITY ANALYSIS

Input. Component FSMs, FSM1 and FSM2; EIPL (external interaction point list), IIPL (internal interaction point list). Assume the following functions: input (T*i*) returns null for a spontaneous T*i* otherwise to the result returned we apply the functions ip and int to get the interaction point and the interaction, respectively; output (T*i*) returns the next output to which the functions ip and int apply the same way as in input (T*i*). The output function returns null when there is no more output left. Similarly, the functions to (T*i*) and from (T*i*) return the state values.
Output. Combined FSM, FSM12, or list of errors.

```
STEP 1.
  For each transition Ti in FSM1 do
  If (input(Ti) = null) or (ip(input(Ti)) in EIPL) then
  repeat
     out1 = output(Ti);
     ifout1 <> null then
      if ip(out1) in IIPL then
      begin
        for each transition Tj in FSM2 do
        if ip(input(Tj)) = ip(out1) then
        begin
          create a combined transition from Ti and Tj;
          tag the combined transition if Tj has any
```

```
        output to an internal interaction point
       end
    end
  until out1 = null;
  For each transition Ti in FSM2 do
  ... similar processing as above ...
  For each tagged combined transition Tij do
    repeat
    out1 = output(Tij);
    ifout1<>null then
     if ip(out1) in IIPL then
     begin
       for each transition Tk in FSM1 or FSM2 do
       if (ip(input(Tk)) = ip(out1)) and (to(Tij) =
       from(Tk)) then
       begin
          to(Tij) := to(Tk);
          write output(Tk) to the output list of Tij
       end
    end
    until out1 = null;
STEP 2.
  For each transition Ti in FSM1 that has no input or
  output with any internal interaction points do
     for state1 in States(FSM2) do
     begin
        add Ti to the list of combined transitions to
        be processed in Step 3 by pairing its from and
        to states with state 1
     end
  For each transition Tk in FSM2 that has no input
  or output with any internal interaction points do
  ... same as above ...

STEP 3.
  StateList := f;
  For each combined transition Tij do
  begin
     StateList := StateList + from(Tij);
     StateList := StateList + to(Tij)
  end;
  For each transition Ti output from Step 2 do
     if the pair (from(Ti), to (Ti)0 in StateList then
        output Ti to the list of combined transitions
     else
        eliminate Ti;
End of the Algorithm.
```