AUTOMATED TESTING OF PROTOCOL SPECIFICATIONS AND THEIR IMPLEMENTATIONS

H. Ural and R.L. Probert
Software Reliability Research Group
University of Ottawa, CSI
Ottawa, Canada K1N 9B4

ABSTRACT

A highly automated approach is proposed for validating the consistency of distinct representations of an identical software functionality. This approach is based on checking whether the observable behaviors presented by software functionality representations given in different levels of abstraction are consistent with one another. These representations are encoded in a relatively mechanical way as procedures in sequential PROLOG, a very high level language. The approach is extremely flexible, straightforward to use, and particularly appropriate to layered protocol architectures. As well, the approach leads to more reliable and less tedious, less error-prone test harness construction and maintenance. A number of variations and extensions of the approach are given. The use of this approach is illustrated in the context of communication protocols design and evaluation.

INTRODUCTION

A communications software development process typically involves a sequence of phases starting from formulating intentions of the desired software functionality and converging to an implementation of the system providing this functionality. In each phase of this process, a representation of the software functionality is developed. Each representation describes the system functionality in terms of externally observable behavior of the system (i.e., possible types and orderings of interactions that the system exchanges with its environment). These representations (i.e., intentions, natural language assertions, formal specifications,

designs, and implementations, etc.) are all distinct but highly related approximations to an identical software functionality, given in different levels of abstraction. Levels of abstraction differ in specifying details of system structure, formats of interaction primitives and their parameters, etc.

Each representation (except intentions) is derived from the representation immediately preceding it. A large number of errors may be introduced into the representations during these derivations (i.e., transformations from one representation to another) mainly due to misunderstandings among project personnel and lack of clarity of the representations as well as omissions, ambiguities, and inconsistencies in the representations themselves. Such errors may propagate throughout subsequent representations unless they are detected and fixed in the representation where they originated.

Thus, the purpose of validation in a communication software development environment is to ensure that the distinct representations of system functionality are in fact consistent. Validation activities in such an environment vary from formal verification and symbolic execution on one hand to simulation and testing on the other.

In this paper we present a validation approach based on checking observable behaviors of software functionality representations given at different levels of abstraction. The consistency of a representation in phase K (representation K) with respect to the representation in phase K-1 (representation K-1) is checked via comparison of their observable behaviors. This approach is somewhat similar to the "trace checking" approach discussed in [JaBo 83], in which a trace checker is semi-automatically constructed from a given representation K-1. The trace

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

checker is then used to check the consistency of the representation K with representation K-1 by evaluating the traces (execution histories) observed during the "execution" of representation K. In our approach, the need for constructing trace checkers from given specifications is eliminated by providing PROLOG procedures implementing the attributed grammar representations which in turn are based on the extended finite state machine formalism. These PROLOG procedures serve more than one function; e.g., the PROLOG procedure implementing representation K can be executed both as a generator of potential traces of representation K+1 (called trajectories) and as a reference for checking traces observed during the execution of representation K+1.

In the following section, we present the basic components of our validation approach which seems to be particularly appropriate to layered protocol architectures.

THE APPROACH

The validation activity that we are dealing with is to check the consistency of a representation of the system functionality with a more abstract representation describing the same functionality. The aim of such an activity is to detect the inconsistencies (if any) between the two representations (e.g., representation K and representation K-1) which are given at different levels of abstraction. This corresponds checking whether a representative set of observed behaviors of representation K is permitted by representation K-1 and a representative set of intended (or expected) behaviors of representation K (derived from representation K-1) is achievable by representation K.

In general, assuming that representation K-1 has already been validated, the following approaches may serve this purpose:

- 1) Apply a selected set of representative sequences of input stimuli to representation K and record its responses together with input stimuli in the order of their occurrences. Each such sequence of input stimuli and responses is called a trace of representation K. (i.e., a sequence of interactions in which representation K participates during its execution). That is, each trace corresponds to an instance of actual behavior of representation K. Since, representation K-1 has already been validated, the system behavior described by representation K-1 can be considered as the expected behavior of representation K. Thus, each trace of representation K can be checked for consistency with the expected behavior described by representation K-1. This validation activity is called "trace checking".
- 2) Obtain a selected set of representative interaction sequences (called trajectories) from representation K-1. This set contains inten-

ded sequences of interactions between representation K and its environment. Then, apply each trajectory to representation K to check whether it is actually allowed. We call this validation activity "trajectory checking".

In each of these cases, discrepancies between observed and expected behaviors of representations are reported for subsequent analysis and reconciliation.

The above validation approaches as well as other similar ones require:

- 1) Construction of "executable" representations of system functionality
- 2) Generation or selection of some representative, managably sized set of sequences of input-stimuli and responses combined or possibly input-stimuli alone .
- 3) Observation and evaluation of interactions between these representations and their environments.

We now describe the implementation of these three steps in some detail.

CONSTRUCTING EXECUTABLE FUNCTIONALITY REPRESENTATIONS

Assuming the system is described in some reasonably precise formalism, [Boch 80, ISO 83], our construction process consists of two steps. In step 1, we construct an attributed grammar representation of the (possibly non-deterministic) system incorporating trace and trajectory selection mechanisms. In step 2, we translate the attributed grammar representation into interpretable PROLOG code and augment this code with invocation procedures. At present, we have developed a prototype translation system to carry out both steps automatically. Thus, our construction of executable representations involves three formalisms; an extended finite state transition language, attributed grammars, and PROLOG (a language for logic programming).

Extended Finite State Transition Formalism

We assume that each system is characterized by the interaction model given in [JaBo 83]. Accordingly, each system is described as a collection of processes, each interacting with its environment (i.e., other processes) through a number of ports. At each port of a process certain types and sequences of interactions are possible. An interaction type is characterized by a name, a number of parameters, and ranges of valid values for those parameters. An occurrence of an interaction involves a particular assignment of valid values to parameters and a transfer of this information to or from the process at a particular port. If an interaction transfers information to the process, we call it an "input" interaction and if it transfers information to the environment we call it an "output" interaction.

We assume that only one interaction can take place at a given port at any given time. In order to identify the interactions occurring at a given port within a given sequence of interactions and to determine their types (i.e., input or output) some attributes such as port number, type, etc. can be attached to interactions.

Moreover, we assume that each process is described by an extended finite state machine where extensions are defined in terms of additional state variables and interaction parameters [Boch 80, ISO 83]. (A similar approach appears to be possible for other formal description techniques [Logr 83]; however, the extended finite state machine approach was selected because of its widespread use). A state transition in the machine is defined by an enabling predicate which involves input interactions, state variables and parameters, and by an action which updates some state variables and possibly generates some output interaction. The next transition to be performed by the process machine is determined by the state of the machine, the parameters of the input interaction (if any), and the truth value of the associated enabling predicate.

It should be noted that some transitions may be spontaneous transitions which can be initiated at any time (i.e., without the reception of an input interaction). As well, at some states there may be more than one transition for a given input interaction (non-determinism in process descriptions is introduced by the existence of such transitions).

In the validation of such non-deterministic representations, the next transition to be performed during execution is specified by augmenting the representation with attributes to control trace or trajectory selection. This produces a corresponding attributed grammar representation.

Attributed Grammar Formalism

Given the system functionality representations in the extended finite state machine formalism, we mechanically translate these specifications into corresponding, equivalent specifications in an attributed grammar formalism. The attributed grammar based specifications may be written in a syntax similar to the one proposed for test sequence grammars in [UrPr 83b]. Each process description given in the extended finite state machine formalism is represented by a production rule in the corresponding attributed grammar formalism. Hence, the transitions defined in process descriptions are represented by alternative terms; each of which may consist of a guard representing the enabling predicate and a computation rule corresponding to the actions taken during the execution of the associated transition. The guards are defined in terms of attributes corresponding to present state, input interactions, contents of channels and buffers, various boolean predicates, etc. The computation rules assign values to some attributes which correspond to updating state variables; preparing

output interactions, etc. The general form of production rules representing process descriptions is as follows:

```
production rule ::= transition { "!" transition }
transition ::= [ guard ":" ] factor { factor }
```

```
factor ::= nonterminal [ "(attributes)" ]
          | terminal [ "(attributes)" ]
          | "!" computation_rule
```

```
where { } stands for repetition (i.e., zero or
more times)
[ ] stands for optional
| stands for alternative
```

Notice that the attributed grammar formalism preserves the non-determinism in the given extended finite state machine representation; all possible transitions at a given state of the process are represented by an enumeration of alternative terms in the grammar. As well, the choice of which alternative term to use when expanding a non-terminal is not determined by the grammar. That is, the grammar merely shows what choices are possible, but it does not specify how this choice is to be made.

However, attributes provide the capability of guiding the choice of alternative terms during the expansion of non-terminals. External to the grammar, one can specify choices of particular non-terminals via the assignment of attribute values. This feature is particularly useful during generation of a selected set of expected behaviors of a representation.

Prolog Encoding of Representations

We implement the attributed grammar representations in the logic programming language PROLOG [Kowa 79, Colm 78]. Figure 1 shows the correspondence in style and semantics between an attributed grammar and its implementation in PROLOG. PROLOG provides a top-down left-to-right expansion of the implemented grammar. That is, the order in which non-terminals are expanded and appropriate alternative terms are considered for the expansion is specified by a sequential top-down left-to-right search and backtracking. In this manner, all possible non-deterministic choices for the expansion of a non-terminal (or all possible transitions in a given state) can be tried one after the other, or users may guide the expansion of non-terminals with the values of attributes.

It is important that PROLOG procedures can be "executed" forwards (from input to output) or backwards (from output to input) as well PROLOG procedures can be invoked by other PROLOG procedures. These features will be utilized in the next section.

```

from(state1=idle and state2=idle)
  when AP1.T_CONNECT_req(parameters)
  provided(no_congestion)
  to(state1=wait_for_acc, state2=idle)
  begin
    caller:=from_T_address
    called:=to_T_address
    options:=requested_options
    buffer12.clear
    buffer21.clear
  end;

```

Figure 1.a) Extended FSM

```

((state1=idle and state2=idle)
and (input-interaction-AP1-id=T_CONNECT_req)
and(no-congestion)):
  !(next-state1=wait_for_acc
  next-state2=idle
  caller=from_T_address
  called=to_T_address
  set options(requested options)
  clear_buffers (buffer12, buffer21))

```

Figure 1.b) Attributed Grammar

```

STATE1='idle', STATE2='idle',

INPUT_1_ID='T_CONNECT_req',
no_congestion,

NEXT_STATE1='wait_for_acc',
NEXT_STATE2='idle',
CALLER =FROM T_ADD,
CALLED =TO T_ADD,
action_1(REQ_OPTIONS),
action_2(BUFFER12, BUFFER21),

```

Figure 1.c) PROLOG Encoding

GENERATION OR SELECTION OF REPRESENTATIVE INTERACTION SEQUENCES

A PROLOG procedure implementing an executable representation may be executed as a generator to produce:

- i) a set of all possible output interaction sequences for a given input interaction sequence
- ii) a set of all possible input interaction sequences for a given output interaction sequence
- iii) a set of all possible interaction sequences satisfying a certain functional property such as the ones corresponding to an unsuccessful transport connection establishment attempt by one of the users.

Exhaustive checking of the consistency of representations is practically infeasible [Piat 80]. A practical compromise is "random" testing which is criticized from viewpoints of complete-

ness [Prob 82] and effectiveness [Howd 82]. Some automated test construction techniques generate somewhat effective test sequences based on system functionality representations given in finite state machine [Chow 78] or grammar [Bazz 82] formalisms. These techniques, however, disregard the values and variations of parameter fields of interactions referred in representations. A more complete set of test sequences may be obtained by augmenting the representations in order to selectively generate representative tests, to encode various testing heuristics, and to incorporate context sensitive information to guide the selection of test sequences.

The augmentation is based on specifying conformance, consistency, or robustness criteria, identifying protocol functions and their functional dependencies, defining representative parameter values and so on. These augmentations are implemented via additional attributes which provide the capability of guiding the test sequence generation process. The augmented attributed grammar based test sequence specifications are then implemented as test sequence generators in PROLOG.

EVALUATING INTERACTION SEQUENCES

The PROLOG procedure implementing representation K may also be used as a validator to validate traces observed during the execution of representation K+1 or to check whether trajectories obtained by executing representation K-1 are allowed by the representation K.

Checking both whether a trace of representation K+1 is valid or whether a trajectory obtained from representation K-1 is allowable is performed by representation K in the same manner. In each case, the interaction sequence is read sequentially from left to right and a match for the interaction sequence is searched. For each input and output interaction all possible states that the implemented process can assume and all possible transitions allowed at those states are investigated. The states and transitions which do not follow from or lead to the interaction being analyzed are not further considered in the analysis. Therefore, all subsequences of interactions which do not lead to a match for the given interaction sequence are eliminated.

The efficiency of the search for a match may be increased by guiding the elimination process via a selected sequence of invocations of the PROLOG procedure. These invocations can be generated by a higher level PROLOG procedure, denoted the invocation-generator, which implements the user-defined elimination strategy. As well, another higher level PROLOG procedure, denoted the test-manager, can be constructed to control and monitor the consistency checking process and to generate discrepancy reports. The effectiveness of such high-level PROLOG procedures has been observed in other work by the authors [Prob 83].

In the following section we sketch the application of our approach to the development and validation of layered communication protocols.

APPLICATION OF THE APPROACH TO LAYERED PROTOCOL DEVELOPMENT

In the layered architecture [Zimm 80], the service provided by a protocol layer N is described in the N-service specification. Here, N-service is specified in terms of possible types and execution sequences of interactions of the layer N with the layer above. In fact, N-service is provided by the collaboration of the entities in layer N and the (N-1) service as shown in Figure 2.

Figure 2: N-Service Provider

A N-protocol entity participates in providing the N-service by communicating with other N-protocol entities via the service of the N-1 layer. The operation of an N-protocol entity is described by an N-protocol specification in terms of its responses to N-Service and (N-1)-service primitives, Protocol Data Units (PDUs), and internally activated actions. In other words, a N-protocol specification is an abstract description of the behavior of an implemented N-protocol entity. Thus, a complete implementation of an N-protocol entity may be constructed by successively refining the N-protocol specification.

Assume that:

- (i) each of the following is represented in terms of an extended finite state machine formalism such as, for example, FDT [ISO 83].
 - a) N-Service Specification
 - b) N-Protocol Specification
 - c) (N-1)-Service Specification
- (ii) implementation (possibly only a prototype) of a N-protocol entity exists.

It is possible to automatically translate the above specifications from an extended finite state machine formalism into corresponding, equivalent specifications in an attributed grammar formalism, and then construct corresponding PROLOG procedures [Ural 83]. These PROLOG implementations of attributed grammar specifications are "executable" and we will refer to them simply as specifications in this section.

Therefore, consistency checking techniques may be applied to validating the consistency of:

- (a) the service specification with respect to the intentions
- (b) the protocol specification with respect to the service specification
- (c) protocol implementations with respect to the protocol specification

A- Validating Service Specifications With Respect to Intentions

Due to the absence of a higher level formal specification, validating the consistency of the N-service specification against the intentions of the specifiers is an informal activity. Two directions can be taken:

- (a) The N-service specification may be executed to yield a representative set of traces. This set can be inspected by the specifier manually; any discrepancies which are detected are analyzed and reconciled. Where necessary the N-service specification can be corrected or augmented.
- (b) The specifier may formulate a small set of representative and discriminating test trajectories which contain selected, intended sequences of interactions between the N-service provider (i.e., N-service specification) and the N-service users. Each trajectory may then be automatically applied to the N-service specification to check that it is indeed allowed by the N-service specification.

B- Validating Protocol Specifications with Respect to the Service Specification

A configuration to check whether a given N-protocol specification is consistent with the N-service specification in the existence of the (N-1)-service specification is given in the following figure:

Figure 3. A Configuration for Consistency Checking

In this configuration, the behavior of the protocol layer (i.e., N. service) is observed by executing two copies of the N-protocol specification interacting through an executable (N-1) service specification. Thus in each of the following scenarios, we are able to validate the participation of the N-protocol specification in the provision of the N-service.

- (i) The N-service specification is executed to generate a selective set of (possibly all) user observable interactions as trajectories. Then these trajectories are applied to the configuration to check whether they are possible in the refined specification which includes the N-protocol specifications and the (N-1) service specification.
- (ii) The configuration is executed to generate the observable interaction sequences as traces. Then these traces are applied to the N-service specification to check whether they are permissible execution histories.
- (iii) The same set of tests are applied to the N-service specification and to the configuration either, one after the other (off-line testing) or to both of them at the same time (on-line testing). In either case, the observed interaction sequences are checked against each other.

Here, the N-protocol specification may be composed of descriptions of behaviors of several interacting modules. A final version of the protocol specification may be derived through a step-wise process involving a series of refinements from a high level of abstraction to a detailed representation. The above configuration may be employed at each step of such a step-wise refinement to validate the design in its most recent form, and to compare the behavior of the most recent design with that of its predecessors.

C- Validating Protocol Implementations With Respect to the Protocol Specification

A given N-protocol entity implementing the N-protocol specification can be checked for conformance to the protocol specification in the following configuration:

Figure 4. A Configuration for Protocol Implementation Testing

In this configuration, the test sequences given in terms of upper and lower interface sti-

mul allowable by the protocol specification are applied to the protocol implementation and its responses are observed at its interfaces. The conformance of the protocol implementation under test to the N-protocol specification is determined by checking that the observed interaction sequences are possible sequences according to the given specification. The incompatibility of the types and parameters of the interactions can be eliminated by using encoder-decoder modules as proposed in [UrPr 83a].

SUMMARY and CONCLUSIONS

We have presented an approach to check the consistency of different representations of identical software functionality with respect to one another. The consistency checking is based on the observable behaviors presented by these representations at different levels of abstraction. These representations may be requirements, functional specifications, detailed designs, or implementations of a given system.

In this approach, the representations (e.g., specifications) given in extended finite state machine formalism are translated into equivalent specifications in attributed grammar formalism. This intermediate translation appears to be necessary. Then, PROLOG procedures are constructed to implement these attributed grammar based specifications. The PROLOG procedures are "executable" and can be used to generate either traces or trajectories from the specifications that they implement. As well, a PROLOG procedure implementing an abstract representation (e.g., design of a system) is used to check whether a trace obtained by observing the execution of a less abstract representation (e.g., implementation of the system) is a permissible execution history or to check whether a trajectory obtained by executing a more abstract representation (e.g., functional specifications of the system) is achievable.

The approach is particularly appropriate to layered protocol architectures. Due to space limitations, we have presented the application of the approach to communication protocols development and validation in general terms. Our experimentation with the approach is focused on the transport protocol development and validation. Some examples and practical results of these experiments are reported in [Ura1 83].

This validation approach is extremely flexible, easy to use, and widely applicable in a software development process which progresses via step-wise refinement.

REFERENCES

- [Bazz 82] Bazzichi, F. and I. Spadafora, "An automatic generator for compiler testing", IEEE Trans. on SE., Vol. SE-8, No. 4, July 1982, pp. 343-353.

- [Boch 80] G.V. Bochmann, "A General Transition Model for Protocols and Communication Services", IEEE Trans. on Communications, Vol. COM 28, 4 (April 1980), pp. 643-650.
- [Chow 78] Chow, T.S., "Testing Software design modelled by finite-state machines", IEEE Trans. on SE., Vol. SE-4, No. 3, May 1978.
- [Colm 78] Colmerauer, A., "Natural language communication with computers", in Lecture Notes in Computer Science (L. Bolc), Springer-Verlag, New York, 1978).
- [Howd 82] Howden W.E., "Weak mutation testing and completeness of test sets", IEEE Trans. on SE., Vol. SE-8, No. 4, July 1982, pp. 371-379.
- [ISO 83] "Tutorial on formal description techniques (FDT)", ISO TC97/SC 16N, May 1983.
- [JaBo 83] Jard, C. and G.V. Bochmann, "An approach to testing specifications", Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Pacific Grove, Cal. March 1983, pp. 53-59.
- [Kowa 79] Kowalski, R., Logic for Problem Solving, New York, North-Holland, 1979.
- [Logr 83] Logrippo, L., "Constructive and executable specification of protocol services by using abstract data types and finite state transducers", Proc. of 3rd Int. Workshop on Protocol Specification, Testing, Verification, Zurich, May 1983, pp. 111-124.
- [Piat 80] Piatkowski, T.F., "Remarks on the feasibility of validating and testing ADCCP implementations", Proc. of Trends and Applications, May 1980, pp. 94-109.
- [Prob 82] Probert, R.L., "Grey-Box (Design-Based) testing techniques", Proc. of 15th Hawaii Int. Conf. on System Sciences, 1982, pp. 94-102.
- [Prob 83] Probert, R.L., D.R. Skuce and H. Ural, "Specification of representative test cases", Proc. of 16th Hawaii Int. Conf. on System Sciences, Jan 1983, pp. 190-196.
- [Ural 83] Ural, H., "A life-cycle approach to testing protocol implementations", Ph.D. thesis (in preparation).
- [UrPr 83a] Ural, H. and R.L. Probert, "Architectures for testing protocol implementations", Proc. of CIPS'83, Ottawa, May 17, 1983, pp. 130-136.
- [UrPr 83b] Ural, H. and R.L. Probert, "User guided test sequence generation", Proc. of 3rd IFIP/WG6.1 Int. Workshop on Protocol Specification, Testing, and Verification, Zurich, May 1983, pp. 421-436.
- [Zimm 80] Zimmermann, H., "OSI Reference Model - The ISO model of Architecture for OSI", IEEE Trans. on Comm., Vol. 28, No. 4, 1980.