Evolutionary Synthesis of Communication Protocols

S. G. Araújo A. C. P. Pedroza A. C. Mesquita

Electrical Engineering Dept.
Federal University of Rio de Janeiro
C.P. 68504 - CEP 21945-970 - Rio de Janeiro - RJ - Brazil
Tel: +55 21 2260-5010 - Fax: +55 21 2290-6626
{granato,aloysio}@gta.ufrj.br mesquita@coe.ufrj.br

Abstract – *A methodology for the synthesis of communication protocols based on evolutionary techniques is presented. It enables to automatically generate finite-state models that synthesize protocol specifications from partial input/output sequences. These partial sequences, directly derived from service specifications and a set of associated protocol data units (PDUs), are the training sequences that drive the evolution of finite-state machines (FSMs), each describing one protocol entity specification. The proposed approach has the advantage of reducing (or even eliminating) computation applicable to finite-state automata (FSA) in existing protocol synthesis methods, abstracting the protocol designer from such tasks.*

1. INTRODUCTION

In a distributed system, multiple autonomous computers provide services to the users acting as a single (virtual) server. Communication protocols are essential components of these systems. A communication protocol is defined as a set of rules governing the format and meaning of the frames, messages or packets that are exchanged by the peer entities (i.e., computers) [1].

Two different abstraction levels are used for specifying distributed systems [2]. At the *service specification* level, protocol entities and communication channels among them are masked. The specification at this level is described as *exchange of primitives* on Service Access Points (SAPs). At the *protocol specification* level, protocol entities actions and messages (and their temporal ordering) are specified. Service and protocol specifications must be equivalent, i.e., they must exhibit the same behavior to the service users.

Protocol synthesis is a central step of the protocol engineering cycle (Fig. 1). It is defined as the generation of protocol specifications from service specifications. A number of strategies for automatically synthesizing correct protocol specifications from given service specifications have been proposed mostly for FSM, LOTOS, Petri net (PN) models and extensions of these models. As remarked in [3], these strategies were concerned mainly in implementing complex control-flows [4], supporting timing constraints [5, 24] and managing distributed resources [6].

Yamaguchi et al. [4] proposed an algorithm that automatically synthesizes a correct protocol entity specification from service specification and a given set of gates and registers in a Petri net model with registers (PNR). The basic idea is to replace each transition of the service specification with a sub-Petri net that simulates the transition. This approach allows the description of structured control-flows as, for example, those containing parallel events.

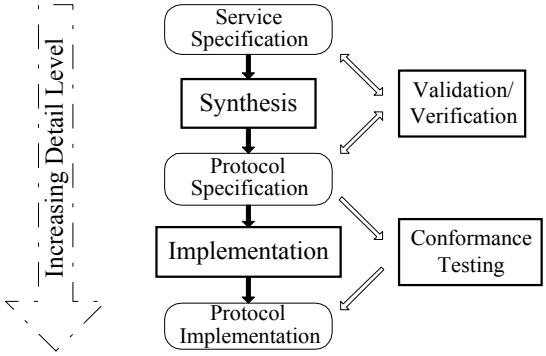

Fig. 1: Protocol engineering cycle

Park & Miller [5] proposed a specification model called timed extended FSM (TEFSM) and presented a method to algorithmically derive a protocol specification from a service specification based on this model. TEFSM can explicitly describe concurrency (allowing multiple simultaneously active states), synchronization and timing requirements such as delay and timeouts. El-Fakih et al. [6] proposed a method to synthesize protocol specification from service specification in PNR concerning in optimizing the communication cost.

On the other hand, Evolutionary Algorithms (EAs), and variations, have been used to explore the solution space for the synthesis of sequential machines. An early attempt to evolve finite-state automata (FSA) was addressed by Fogel [7, 8] through Evolutionary Programming (EP). This approach uses some of the EAs concepts such as population, random initialization, generation, mutation and reproduction (cloning) to evolve an automaton that predicts outputs based on known input sequences. However, EP has shown poor performance by the lack of the crossover genetic operator [9, 10].

Recent works have been successful in synthesizing sequential systems with the aid of Genetic Algorithms (GAs) [11, 12, 13]. Ref. [11] presented an approach to synthesize

0-7803-7661-7/03/$17.00©2003 IEEE.

synchronous sequential logic circuit from partial input/output sequences. By using *technology-based* representation (a netlist of gates and flip-flops) the method was able to synthesize various types of small FSMs, such as serial adder and 4-bit sequence detector. Others approaches [12, 13] used *state-based* representation in which the next-state and the output corresponding to each current state-input pair of the state-transition table are coded in a binary string defined as the *chromosome*. Since, in this case, the number of states of the FSA is unknown, a large number of states must be used at the start.

Although state-based representation does not have the intrinsic drawbacks of the technology-based representation, it is not efficient for systems with large number of states since it provides a complete temporal ordering of all potential distinctions leading to a large, complex behavior description [14]. However, the benefit of using it in the protocol synthesis is that it guarantees the completeness of the synthesized specifications.

In [15] formal grammars are exploited as alternative to evolve FSA. In this approach, hardware behaviors are automatically generated as Hardware Description Language (HDL). The authors used Genetic Programming (GP) to evolve trees from productions in order to create HDL-descriptions. The paper discussion is mainly concerned with HDL, however, this technique is applicable to ordinary computer programs written in conventional languages such as 'C'.

In this work the protocol synthesis problem is approached through the use of evolutionary techniques. Both service and protocol specifications are modeled by FSMs. The main point addressed is the use of GP with a multi-parameter fitness function to automatically synthesize protocol descriptions from service specifications. Section 2 gives some basic definitions of FSM and a service specification example using this model. Concepts of Evolutionary Computation are also introduced. Section 3 describes the synthesis methodology. In section 4 a connection-oriented protocol is used as example to test the performance of the proposed approach.

2. BASIC DEFINITIONS

2.1. FSM and Service Specification

A finite-state machine (FSM) is commonly used for specifying communication protocols due to its simplicity and precise definition of the temporal ordering of interactions [16, 17]. From the two traditional Moore/Mealy FSMs models it is found that in the Mealy machines, since the outputs are associated with the transitions, some behaviors can be implemented with fewer states than Moore machines.

The Mealy FSM model M is formally defined as a 7-tuple $\{Q,V,t,q_0,V',o,D\}$ where $Q \neq \emptyset$ is a finite set of states, V is a finite input alphabet, t is the state transition function, $q_0 \in Q$ is the initial state, V' is a finite output alphabet, o is the output function and D is the specification domain, which is a subset of $Q \times V$. t and o together characterize the behavior of the FSM, i.e., $t(q,v)$: $Q \times V \rightarrow Q$ and $o(q,v)$: $Q \times V \rightarrow V'$. If $D = Q \times V$, then t and o are defined for all possible state/input combinations and therefore the FSM is said to be *completely specified*. Otherwise, it is a *partially specified* FSM.

The service provided by a communication protocol is specified by a set of *service primitives* (simply called *primitives*) available to the user to access the service. Fig. 2a) gives the *service specification* (*SS*), in FSM model, of a connection-oriented protocol. Figs. 2b) and 2c) give the (same) *SS*, from the site perspective (sender and receiver). Note: (i) primitive A_i occurs in site i; (ii) a sending (to the user) and a receiving (from the user) of the primitive A is denoted !A and ?A, respectively. The connection is initiated by the sender (*C_Req*) and the receiver may either accept (*C_Resp*) or reject it (*D_Req*). If the service is established (state 3 in Figs. 2b) and 2c)), the sender can call for the reinitialization procedure (*R_Req*) in order to remove all data from the medium. Finally, the connection can be disconnected by the sender (or the receiver) using *D_Req* primitive. Data transfer primitives were omitted for simplicity.

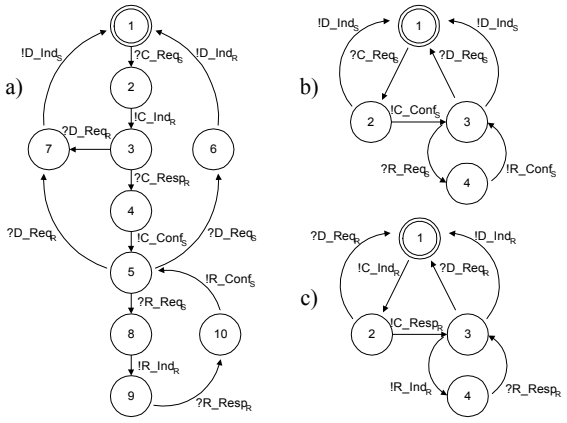

Fig. 2: a) Global *SS*, b) *SS* of the sender and c) *SS* of the receiver

2.2. Evolutionary Computation

Evolutionary Computation (EC) consists of a class of probabilistic algorithms inspired by the principles of Darwinian natural selection and variations that can greatly benefit from the increased simulation power for complex design, control and knowledge discovery applications [18]. Examples of EC are Genetic Algorithm (GA), Classifier Systems (CFS), Genetic Programming (GP) and Artificial Life (a-life).

GA [19] emulates the way nature improves the traits of living beings. The *chromosome* is the basic component of the GA. It represents a point, called in this context an individual, of the search (or solution) space of the problem. The fitness value measures how close the individual is to the solution. By iteratively applying the genetic operators (fitness evaluation, fitness-based selection, reproduction, crossover and mutation) to a randomly started population

of individuals, such population evolves in order to breed at least one offspring with the desired behavior.

GA usually works with strings (chromosomes) of *fixed length* in which one or more parameters are coded. GA solutions are best applied to poorly understood or highly metastatic problems for which precise, deterministic solutions are not available. GP [10] is a branch of GA, the main difference being the solution representation. Unlike GA, GP can easily code chromosomes of *variable length*. This increases the capacity in structure creation allowing more complex coding.

An alternative way to encode domain-knowledge is by formal grammars [20]. Programs may be produced by combining context-free grammar (CFG) with GP [21, 22], what is known as G^3P (Grammar-Guided Genetic Programming). Genetic Programming Kernel (GPK) [22] was used as the core engine for the GP system used in this paper. GPK is a complex G^3P system that evolves programs (or structures) in any language (or notation) once a Backus-Naur form (BNF) is provided as input.

3. METHODOLOGY

The whole methodology is based on the use of the GP algorithm. Rather than create a global system of states and project it into the alphabet of events of each site, as in [23, 24], the system generates each protocol entity (*PE*) separately. The idea is to evolve a FSM that describes the protocol entity specification from partial input/output sequences, in the following called *training sequences*. The execution flow of the proposed methodology is drawn in Fig. 3. In this figure, the GP box outputs a population of FSMs for fitness evaluation. If at least one individual (protocol description) reproduces the I/O behavior given by the training sequences, the system ends outputting the protocol entity specification of site i (PS_i).

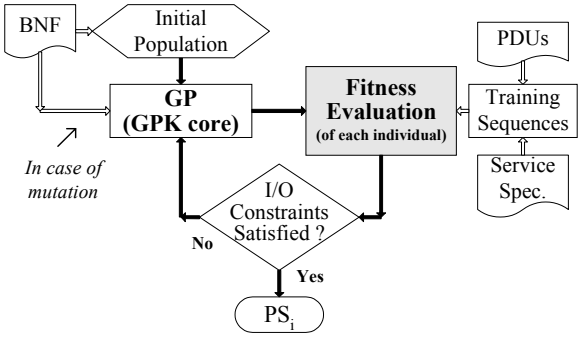

Fig. 3: Execution flow of the methodology

Here, the focus is on the control structure of the protocol. Parameters and data values are not considered. The evaluation of the fitness (shaded box in Fig. 3) is a key procedure of the GP system. The model adopted for the fitness evaluation resembles the well-known *black-box* approach used in protocol conformance testing, as shown in Fig. 4. By using an event-driven interface, the *entity to be evolved* communicates with the service user through primitives and with the remote peer entity through PDUs. It is assumed that the communication system has reliable FIFO channels (i.e., there is no faulty condition). In each evolutionary step (generation) the system probes a population of *entities to be evolved* (the individuals) with input sequences (In_0 and In_1) and records the corresponding output sequences (Out_0 and Out_1). These output sequences are compared with the *correct* (desired) output sequences (the outputs of the training sequences) and a fitness value is assigned to each candidate solution.

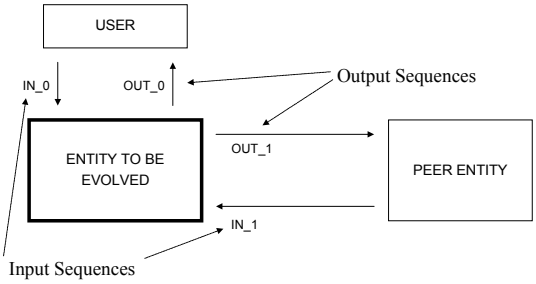

Fig. 4: Model for the fitness evaluation

3.1. Derivation of Training Sequences

A *training sequence TS* is a sequence of *correct* input/output pairs $< v_1/v_1', v_2/v_2', \ldots, v_L/v_L' >$, where $v \in V$, $v' \in V'$ and L is the length of the sequence. Besides the (1) *SS*, assumed sequential (i.e., there is a relation of causality between any pair of consecutive primitives) and deadlock- and livelock-free, the following *protocol elements* must be defined for the complete derivation of a *TS*:

(2) Protocol assumptions and
(3) PDUs and their respective association to primitives.

A connection-oriented protocol is used as example. In the following, PE_s denotes the sender-side entity of the connection-oriented protocol, SS_s denotes the sender-side of the *SS* and PS_s denotes the protocol entity specification for the PE_s. **(1)** The *SS* is shown in Fig. 2a). **(2)** Protocol assumptions are: **(2.1)** each state of a PE_s must be able to respond to every input, i.e., the result of the synthesis process is a completely specified FSM, **(2.2)** only establishment, reinitialization and release phase are employed and **(2.3)** only the PE_s may initiate the connection and request the reinitialization procedure. **(3)** The PDUs, together with the primitives employed in the SS_s (Fig. 2b)), are listed on Table 1, which defines the input alphabet V={0,1,2,3,4,5} and the output alphabet V'={0,1,2,3,4,5,6} of the PS_s. **(3.1)** The primitive-PDU association is defined by the set {*C_Req*$_S$*-con_req*, *C_Resp*$_R$*-con_acc*, *D_Req*$_S$*-dis_ind*, *D_Req*$_R$*-dis_ind*, *R_Req*$_S$*-rei_req*, *R_Resp*$_R$*-rei_acc*}. For example, a *con_req* PDU is sent from PE_s to PE_r (receiver site) after an occurrence of a *C_Req*$_S$ primitive.

Table 1: PDUs and primitives of the PS_s

Input (V)	Description	Type	Coding
C_Req	Connect_Request	Primitive	0
D_Req	Disconnect_Request	Primitive	1
con_acc	connect_accept	PDU	2
dis_ind	disconnect_indication [a]	PDU	3
R_Req	Reinitialization_Request	Primitive	4
rei_acc	reinitialization_accept	PDU	5
Output (V')	**Description**	**Type**	**Coding**
C_Conf	Connect_Confirm	Primitive	0
D_Ind	Disconnect_Indication	Primitive	1
con_req	connect_request	PDU	2
dis_ind	disconnect_indication	PDU	3
R_Conf	Reinitialization_Confirm	Primitive	4
rei_req	reinitialization_request	PDU	5
Null	"No output"	-	6

[a] Used as negative response to the con_req PDU

Derivation Method:

Step_1: A sequence of primitives is created from (global) *SS* (Fig. 2a). In this step it is desired that all paths of the *SS* be present (for this purpose any automatic test sequence generator can be used, generating correct sequences from the SS).

Example: By traversing the states 1, 2, 3, 4, 5, 7, 1, 2, 3, 7, 1, 2, 3, 4, 5, 8, 9, 10, 5 and 6 of the *SS* (Fig. 2a), the following sequence of primitives is obtained:

<C_Req_S, C_Ind_R, C_Resp_R, C_Conf_S, D_Req_R, D_Ind_S, C_Req_S, C_Ind_R, D_Req_R, D_Ind_S, C_Req_S, C_Ind_R, C_Resp_R, C_Conf_S, R_Req_S, R_Ind_R, R_Resp_R, R_Conf_S, D_Req_S, D_Ind_R>.

Step_2: The associated PDU is inserted between any pair of consecutive primitives occurring at different sites, except when a choice between downward primitives is made (by the user).

Example: From the above-defined primitive-PDU association (def. (3.1)), the PDUs are inserted into the sequence:

<C_Req_S, **con_req**, C_Ind_R, C_Resp_R, **con_acc**, C_Conf_S, D_Req_R, **dis_ind**, D_Ind_S, C_Req_S, **con_req**, C_Ind_R, D_Req_R, **dis_ind**, D_Ind_S C_Req_S, **con_req**, C_Ind_R, C_Resp_R, **con_acc**, C_Conf_S, R_Req_S, **rei_req**, R_Ind_R, R_Resp_R, **rei_acc**, R_Conf_S, D_Req_S, **dis_ind**, D_Ind_R>.

Step_3: A new sequence of input/output pairs is obtained after projecting the above sequence into the input alphabet (V) of the PS_s.

Example: Considering the input alphabet defined on Table 1, i.e., V = {C_Req, D_Req, con_acc, dis_ind, R_Req, rei_acc}:

<C_Req_S/con_req, con_acc/C_Conf_S, dis_ind/D_Ind_S, C_Req_S/con_req, dis_ind/D_Ind_S, C_Req_S/con_req, con_acc/C_Conf_S, R_Req_S/rei_req, rei_acc/R_Conf_S, D_Req_S/dis_ind>.

Step_4: In order to agree with assumption (2.1) the sequence must include non-expected input events (chosen at random from V – expected input event). Such events must be inserted randomly in the sequence in order to drive the evolutionary process. In this case, the PE_s answers with "Null" output. An exception is made regarding the *dis_ind* and D_Req_S events. In such case, the PE_s must answer with D_Ind_S and *dis_ind*, respectively, and the subsequent event must be C_Req_S primitive in order to match the *SS* (note, in this case, the remainder of the sequence must be re-written).

Example: A *TS* is obtained after this step:

<C_Req_S/con_req, **C_Req_S/Null**, con_acc/C_Conf_S, **rei_acc/Null**, dis_ind/D_Ind_S, **con_acc/Null**, C_Req_S/con_req, dis_ind/D_Ind_S, **R_Req_S/Null**, C_Req_S/con_req, **dis_ind/D_Ind_S**, C_Req_S/con_req, **C_Req_S/Null**, con_acc/C_Conf_S, **con_acc/Null**, **D_Req_S/dis_ind**>.

By using Table 1 coding, the above-obtained training sequence *TS* is showed below:

TS = <0/2, 0/6, 2/0, 5/6, 3/1, 2/6, 0/2, 3/1, 4/6, 0/2, 3/1, 0/2, 0/6, 2/0, 2/6, 1/3>.

3.2. Length of the Training Sequences

The *TS* must be long enough to exercise all paths of the FSM that describes the PS_s. A too short sequence may cause ambiguity in describing the desired behavior. On the other hand, an outsized sequence would affect the system performance.

The solution for the problem of finding the proper length of the input sequences was taken from the formula derived in [11], based in the *waiting times in sampling* problem solution [25]. This formula defines the length L of the input sequence as $L = E(S) \times E(I)$, where $E(S)$ and $E(I)$ are the *expected number of state transitions* and *expected number of inputs*, respectively. Note that $E(N)$ can be computed using $E(N) = N\,(1 + {}^1/_2 + \ldots + {}^1/_N)$. However, since the number of states S required to describe the PS_s is unknown, it must be overestimated a priori.

3.3. Chromosome Coding

The chromosome, which encodes a FSM, uses a state-based representation (SBR) as described in Section 1. The resulting string (chromosome) with S states and I inputs is shown in Fig. 5.

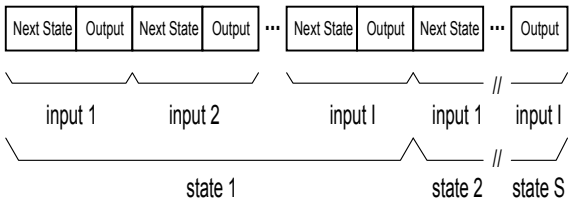

Fig. 5: Chromosome coding using SBR

Unlike others approaches [12, 13], in the present approach the length of the chromosome is allowed to vary. To this purpose a GP algorithm is used, since this is the most efficient way to implement an evolutionary process with variable length chromosomes known to date. Addi-

tionally, the fitness function is weighted by the number of states of the FSM, which induces the evolutionary process to search for FSMs with reduced number of states (registers). As a consequence of the crossover operation between different size chromosomes, the process may lead to unfeasible FSMs. Nevertheless, the fitness evaluation interprets such cases as self-loop, reacting with "Null" output.

3.4. Fitness Function

The fitness function is a multi-parameter function that the algorithm will maximize. The fitness value assigned to a given FSM behavior, here evaluated through an input/output sequence perspective, is weighted by the inverse of the number of states of the FSM. The fitness function F is defined as:

$$F = \sum_{i=1}^{N} w_i H_i + \frac{W}{S} \quad (1)$$

where w_i is a weighting factor for fitness case i, S is the number of states of the FSM, W is a constant, N is the number of fitness cases (*TSs*) and H_i is the number of *output hits* due to the fitness case i (TS_i). H_i is evaluated as follows. Initially, the FSM must be in the reset state. In the sequence, for each input of the TS_i, its output is compared with the *correct* output (the corresponding output of the TS_i) and an output hit is signed in case of a match.

4. EXPERIMENT

In this section the performance of the proposed methodology is tested. The objective is to derive the protocol entity specification (PS_s) for the sender-side entity of the connection-oriented protocol (PE_s) from the training sequences (*TSs*) generated using the derivation method presented in Subsection 3.1.

4.1. BNF Definition

A BNF grammar G describes admissible structures of a language through a 4-tuple $\{S,N,T,P\}$ where S denotes the *start* symbol, N the set of *non-terminal* symbols, T the set of *terminal* symbols and P the *productions*, i.e., rewriting rules that map the elements of N to T. The BNF that allows the variable length state-based chromosome coding is defined as:

```
S          := <fe>;
<fe>       := "S" <expr> "E";
<expr>     := <if_stat>|<expr> <if_stat>;
<if_stat>  := <next_st><out> <next_st><out>
              <next_st><out> <next_st><out>
              <next_st><out> <next_st><out>;
<next_st>  := "0"|"1"|"2"|"3"|"4"|"5";
<out>      := "0"|"1"|"2"|"3"|"4"|"5"|"6";
```

Note that <*expr*> allows the chromosome to have variable length, while <*if_stat*> fixes the number of input events (six, in the above BNF definition, each one yielding a next-state/output pair, i.e., <*next_st*> <*out*>).

4.2. GP Parameters

Table 2 shows the main control parameters of the GP.

Table 2: GP tableau

Objective:	Find a FSM, with reduced number of states, which correctly describes the PS_s from *TSs*.
Terminal symbols:	Outputs: 0, 1, 2, 3, 4, 5 and 6; state identifiers: 0, 1, 2, 3, 4, 5 and 6; string (chromosome) delimiters: "S" (Start) and "E" (End).
Non-terminal symbols:	<fe>, <expr>, <if_stat>, <next_st> and <out>.
Fitness cases:	Eighteen 32 bits *TSs* derived from the *SS* and a set of associated PDUs (see Subsection 3.1).
Raw fitness:	The number of output hits of the eighteen *TSs*, weighted by the inverse of the number of states used to implement the FSM (see equation (1)).
Selection Method:	Fitness-proportionate.
Main GP Parameters:	M=1000, G=400, p_r=0.12, p_c=0.63 (two-point crossover), p_m= 0.25, without elitism.

4.3. Results

Experiments were carried out with a population size (M) of 1000 individuals evolving up to a maximum of 400 generations. The *TS* length was evaluated using six inputs (see Table 1) and an estimated value of five for S, leading to a 168-input/output *TS* (see Subsection 3.2). Moreover, it is desirable to have multiple *TSs* to improve the performance of the learning system (see Ref. [13]). In fact, eighteen *TSs* were used, each with 32 bits in length, which corresponds to more than three 168-length sequences. W was set to 60. w_i was set to 1 for all i since the *TSs* have the same length.

The experiment was done with 8 independent runs. In one particular run, a single individual attained the perfect score of output hits, using a minimum number of states. The resulting Mealy FSM, which successfully describes the PS_s, is given using state-transition graph (STG) in Fig. 6 (note: label v/v' represents an edge e_{ij} between two states q_i and q_j iff $o(q_i,v)=q_j$ and $t(q_i,v)=v'$). This individual was synthesized with just four states in the 119th generation. The evolution of this best individual is shown in Fig. 7. Its main fitness was F = 591 for a total of 18 × 32 = 576 output hits. 100% correct individuals with 5, 6 and 7 states have been noticed in other runs. Figs. 8 and 9 show two such correct but not optimal solutions with respective chromosome coding.

Although evolutionary techniques are intrinsically computational intensive, the system showed good performance. Each of the runs necessary to synthesize one entity of the above discussed experiment consumed (in average) no more than one minute in a 1GHz Pentium-based machine.

Semantic Correctness Discussion

A priori, semantic correctness, which means that the protocol performs its intended functions with respect to the service specification, cannot be guaranteed by FSAs that conform to *partial* input/output sequences (the training sequences). Nevertheless, the present work considers the work of Manovit et al. [11] and Chongstitvatana & Aporntewan [13], which investigated the influence of the length of the sequences as well as the quantity of such sequences for what they called the *correctness percentage*. The correctness percentage was defined as the relation between the number of runs yielding *complete solutions*, i.e., a solution that operates correctly for *all possible* input/output sequences, and the number of runs yielding solutions, i.e., solutions that operate correctly for the *tested* input/output sequences. They showed by experiment that the correctness percentage could be raised to 100% (in this case assuring the semantic correctness) by increasing the length of the sequences and the number of such sequences used in the evolutionary process.

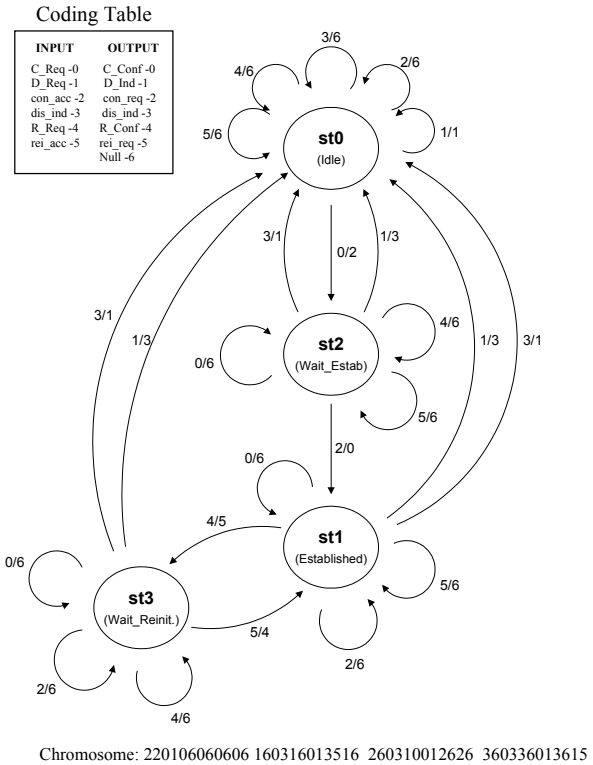

Fig. 6: PS_s, using STG, of the fittest individual

The sequence length and the number of the sequences used in the present experiment were chosen in order to ensure the semantic correctness. As consequence, the experiment succeeded to the equivalence check, for which the Concurrency Workbench (CWB) tool [26] was used. For this purpose, the receiver site was also evolved and both entities (sender and receiver), together with the global service specification (Fig. 2a)), were converted to CCS.

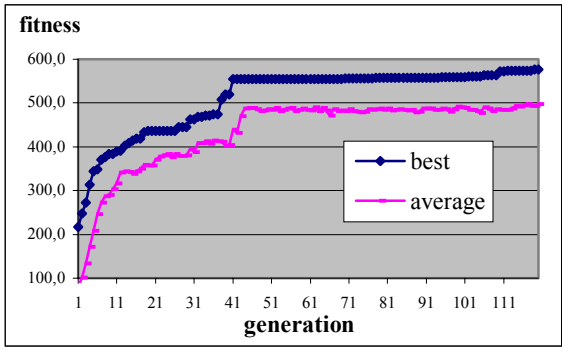

Fig. 7: Best individual fitness x generation curve

5. CONCLUSION

A communication protocols synthesis methodology using evolutionary techniques is proposed. The methodology has as distinctive characteristic the use of Genetic Programming to implement an evolutionary process driven by *training sequences*. These training sequences are derived from the service specifications and a set of PDUs in a straightforward way.

The proposed methodology intrinsically ensures the completeness and the determinization of the protocol specification since it uses state-based representation, which generates a deterministic, completely specified FSM. Semantic correctness is assured by increasing the number and the length of the training sequences (see semantic correctness discussion in Subsection 4.3). Nevertheless, since evolutionary techniques use probability, a more specific study has to be done to quantify the two parameters (sequence length and quantity of sequences) involved in such correctness.

The proposed approach has the advantage of reducing (or even eliminating) some computation applicable to FSA in existing protocol synthesis methods [23, 24], as follows: (i) projection is done in a more direct way, (ii) determinization is eliminated and (iii) FSA is automatically reduced as a consequence of the variable size chromosome evolutionary process together with a fitness function weighted by the number of states. Such advantages enable the protocol engineer to consider the protocol design in a higher abstraction level.

The system showed good performance, as remarked in Subsection 4.3. This encourages the synthesis of protocols with higher number of states and input events, i.e., those of practical interest.

REFERENCES

[1] A. Tanenbaum, *Computer Networks*, 3rd edition, pp. 27, 1996.

[2] H. Yamaguchi, *Implementation of Service Specifications on Distributed Computing Systems*, PhD thesis, Osaka University, 1998.

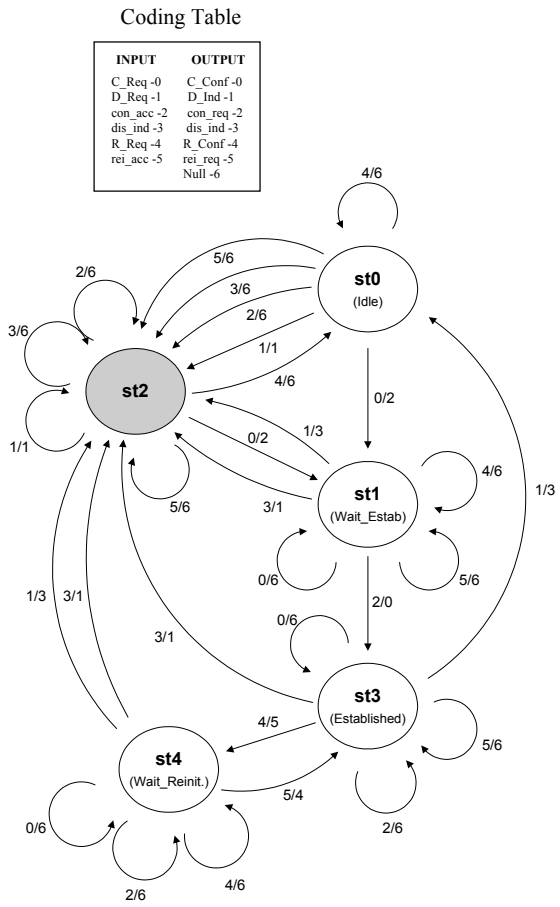

Fig. 8: Correct solution with one redundant state (shaded ball)

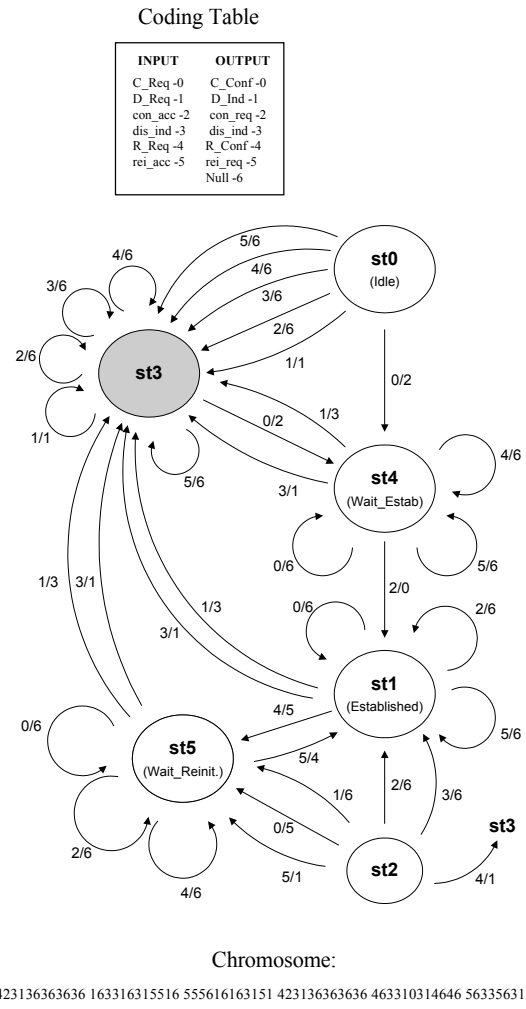

Fig. 9: Correct solution with one unreachable state (small ball) and one redundant state (shaded ball)

[3] K. El-Fakih, H. Yamaguchi, G. Bochmann and T. Higashino, *Protocol Re-synthesis Based on Extended Petri Nets*, in Proc. of Int. Workshop on Software Eng. and Petri Nets (SEPN-2000), pp. 173-188, 2000.

[4] H. Yamaguchi, K. Okano, T. Higashino and K. Taniguchi, *Synthesis of Protocol Entities' Specifications from Service Specifications in a Petri Net Model with Registers*, in Proc. of ICDCS-15, pp. 510-517, 1995.

[5] J-C. Park and R. Miller, *Synthesizing Protocol Specifications from Service Specifications in Timed Extended Finite State Machines*, in Proc. of 17th Int. Conf. on Distributed Computing Systems, 1996.

[6] K. El-Fakih, H. Yamaguchi and G. Bochmann, *A Method and a Genetic Algorithm for Deriving Protocols for Distributed Applications with Minimum Communication Cost*, in Proc. of the 11th IASTED Int. Conf. on Parallel and Distributed Computing and Systems, (PDCS'99), 1999.

[7] L. Fogel, *Autonomous Automata*, Industrial Research, 4:14-19, 1962.

[8] L. Fogel, A. Owens and M. Walsh, *Artificial Intelligence Through Simulated Evolution*, John Wiley & Sons, New York, 1966.

[9] D. Golberg, *Genetic Algorithm in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[10] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[11] C. Manovit, C. Aporntewan and P. Chongstitvatana, *Synthesis of Synchronous Sequential Logic Circuits from Partial Input/Output Sequences*, in Proc. of Int. Conf. of Evolvable Systems (ICES'98), pp. 98-105, 1998.

[12] R. Collins and D. Jefferson, *Representation for Artificial Organisms*, in Proc. of the 1st Int. Conf. on Simulation of Adaptive Behavior, MIT Press, 1991.

[13] P. Chongstitvatana and C. Aporntewan, *Improving Correctness of Finite-State Machine Synthesis from Multiple Partial Input/Output Sequences*, in Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware, pp. 262-266, 1999.

[14] D. Clancy and B. Kuipers, *Model Decomposition and Simulation: A Component Based Qualitative Simulation Algorithm*, in Proc. of the 14th National Conf. on Artificial Intelligence, AAAI Press, 1997.

[15] H. Hemmi, J. Mizoguchi and K. Shimohara, *Evolving Large Scale Digital Circuits*, Artificial Life V, pp. 53-58, 1995.

[16] G. Bockmann and A. Petrenko, *Protocol Testing: A Review of Methods and Relevance for Software Testing,* ISSTA'94, ACM Int. Symposium on Software Testing and Analysis, Seattle, U.S.A., pp. 109-124, 1994.

[17] K. Tai and Y. Young, *Synchronizable Test Sequences of Finite State Machines,* Computer Networks and ISDN Systems, 30:1111-1134, 1995.

[18] J. Rosca, *Hierarchical Learning with Procedural Abstractions Mechanisms*, PhD Thesis, University of Rochester, New York, 1997.

[19] J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan, 1st edition, 1975.

[20] A. Ratle and M. Sebag, *Genetic Programming and Domain Knowledge: Beyond the Limitations of Grammar-Guided Machine Discovery,* Parallel Problem Solving from Nature (PPSN-VI), pp. 211-220, 2000.

[21] P. Whigham, *Grammatically-Based Genetic Programming*, in Proc. of the Workshop on Genetic Programming: From the Theory to Real-World Applications, pp. 33-41, Morgan Kaufman, 1995.

[22] H. Hörner, *A C++ Class Library for Genetic Programming*, Release 1.0 Operating Instructions, Viena University of Economy, 1996.

[23] A. Khoumsi, and K. Saleh, *Two Formal Methods for the Synthesis of Discrete Event Systems*, Computer Networks and ISDN Systems, Vol. 29, No. 7, pp. 759-780, 1997.

[24] A. Khoumsi, G. Bochmann and R. Dssouli, *Protocol Synthesis for Real-Time Applications*, Joint Int. Conf. on Protocol Specification, Testing and Verification (PSTV) and FORTE, Beijing, China, 1999.

[25] W. Feller, *An Introduction to Probability Theory and its Applications*, Vol. I, Wiley, pp. 224-225, 1968.

[26] R. Cleaveland, J. Parrow and B. Steffen, *The Concurrency Workbench: A Semantics Based Toll for the Verification of Concurrent Systems*, ACM Transactions on Programming Languages and Systems, 15(1):36-72, 1994.