Automated test suite generation from formal protocol specification

GÁBOR VINCZE
Budapest University of Technology and Economics, Dept. of Telecommunications and Telematics
vincze@alpha.ttt.bme.hu

***Keywords:* conformance testing, test generation, mutation analysis, evolutionary algorithms, bacterial algorithm**

In this paper, we present a method for automatic test generation from the formal SDL specification of a protocol. Protocol testing is an important step in the development process, but the creation of test suites is a time consuming task. Automating this phase reduces the time necessary for implementation, and cuts an important error source. We show how Mutation Analysis can be used to match test criteria and test cases obtained with a graph exploration algorithm applied on the SDL description of the system. We then use evolutionary algorithms to select an optimal subset from this initial set of test cases. Using these methods, we build a complete process for the automated generation of a test suite from the formal specification of a protocol.

As telecommunication companies had to offer more of services every day, while trying to integrate their networks, telecommunication protocols became increasingly complex. At the same time, the reliability of these networks had to meet ever-higher standards as well.

With this increase in complexity, the effort needed for the specification of protocols became a serious burden, and the need for reliability and interoperability between manufacturers called for more extensive testing. These problems gave birth to formal specification methods, and formal testing methods to verify if implementations behaved according to the specifications.

The most widely used formal languages in the world of telecommunications are the Specification and Description Language (SDL, [1]) for system specification, which models a system as parallel Communicating Finite State Machines (CEFSM), and Tree and Tabular Combined Notation (TTCN, [2]) for black-box test description. Today highly integrated and widely used development tools [3] exist to aid designers in the specification and testing process. However, the creation of a formal test suite still requires considerable effort, and the human factor remains the most expensive and error-prone component in the process. As these tests often have to be run several hundreds or thousands of times, execution time and hardware requirements are also a crucial factor.

In this paper, we present a method for automatic test generation from the SDL description of a system. The test generation process has four main steps:

1) formal specification in SDL
2) creation of a set of test cases by a state-space exploration algorithm
3) mutation analysis
4) selection of an optimal subset of test cases

We will first explain the mutation analysis method in detail; then, we will show how we use evolutionary algorithms to select an optimal subset of test cases from the resulting set, and finally, we will illustrate the whole test generation process by an example on the INRES protocol.

1. Mutation analysis

1.1. Overview

Mutation analysis is a white-box test case development method, which means we possess knowledge on the internal working of the system. Traditional mutation analysis has been developed to find errors in program code, but we use it here on formal protocol specifications instead to select the appropriate black box test cases.

In a mutation analysis system, we need to define a set of mutation operators [4], where each operator represents an atomic syntactical modification. The use of these operators is convenient for two reasons: They allow the formal description of error types, and they allow the automatic generation of mutants. By applying systematically the operators on the specification, we can generate a set of mutants.

A mutation analysis system is made up of three basic components:

- The original system;
- The mutant system, which contains a small syntactical modification compared to the original system. Mutants are obtained by applying the mutation operators, each operator representing a small syntactical modification;
- An oracle – a human or, in most cases, a program, which differentiates the original system from the mutant by observing its interactions with the environment.

We assume that the original CEFSM specification is close to the requirements, and thus test cases detecting syntactical modifications of the specification are useful.

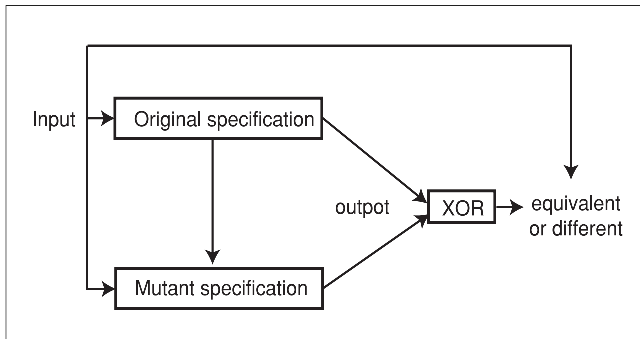


Fig. 1. Mutation analysis

We only produce first-order faults – we apply one mutation at a time – because test cases detecting simple modifications will also detect complex modifications created as a sequence of simple modifications [5]

Test cases distinguish a mutant from the original if it gives a different output. However, part of the mutants generated by the operators might be semantically equivalent to the original system: they give exactly the same output on all possible inputs. We call these mutants *equivalents*. We call *pseudo-equivalents* mutants that are semantically different from the original system, but give exactly the same output on all possible inputs. We should ignore all equivalents during testing, but should consider all non-equivalents during test case selection. This creates a serious problem in mutation analysis, since it is generally not possible to automatically identify equivalents, and the distinction between equivalents and non-equivalents needs human interaction.

1.2. Mutation operators

It is a very important consideration that mutation operators do not create pseudo-equivalents, and minimize the number of equivalents. The basic principles of mutation operator definition are:

- Operators should model atomic faults;
- They should only create first order mutants;
- We should only generate syntactically correct mutants;
- To allow test case generation, we should only create semantically correct mutants;
- Operators should generate a finite, and the lowest possible number of mutants.

Five classes of operators have been defined [4] for CEFSMs, depending on the part of the CEFSM they modify:

- state, input, output, action, and predicate modifying operators.

For each class, we can give three types of operators depending on the type of fault they represent:

- augmenting, reducing and exchanging operators.

1.3. Test case – test criteria matching

The following algorithm allows us to assign a set of test criteria to each test case of a finite sized, unstructured, and highly redundant test suite, which we can obtain for example by a state-space exploring algorithm exploring the system specification. If we apply the mutation operators to observe inopportune inputs, this initial test suite must also contain inopportune test cases.

Let C be a two dimensional matrix of Boolean values.

0) Generate a set of test cases;
1) Apply a mutation operator on the CEFSM to create the ith mutant;
2) Run all the test cases on the mutant specification, and observe inconsistencies: if the test case gives a different result from the original specification, the test case detects the given mutant;
3) Create column vector *Ci* (i^{th} column of the C matrix)
   - Let *Ci[j] = 0* if the jth test case cannot detect the ith mutant;
   - Let *Ci[j] = 1* if the jth test case detects the ith mutant;
4) Repeat steps 2-4. where i goes from 1 to N, where N is the number of all the possible mutants;
5) Acquire the C matrix of criteria, where rows represent test cases of the original set, and columns represent the mutants.

2. Test selection with evolutionary algorithms

The aim of the selection process is to obtain an optimal subset of test cases from an already existing unstructured, highly redundant set. To achieve this goal, we applied three different soft algorithms: the Genetic Algorithm (GA), the Pseudo-Bacterial Genetic Algorithm (PBGA) and the Bacterial Evolutionary Algorithm (BEA).

We chose to use evolutionary algorithms for test selection because they provide high quality solutions in acceptable time, can handle very complex cases, and can be easily integrated into the test generation process [6].

2.1. General considerations

Individuals: An *individual* is a possible solution of the problem, an optimized test suite in our case. We had two different ways of representing test suites: either a fixed length string of N bits, where N is the number of all the test cases in our original set, each bit's value being 1 if the test suite includes the corresponding test case (which we called *bit-string* individuals), or a variable size set of values between 1 and N, where each number represents the number of a test case in the original set (which we called *pointer-set* individuals).

In the latter case, it is of course possible to have test suites that incorporate the same test cases twice, but these will have an increased execution cost without any added value and will be eliminated during the selection process. We used one or both representation methods, depending on the algorithm.

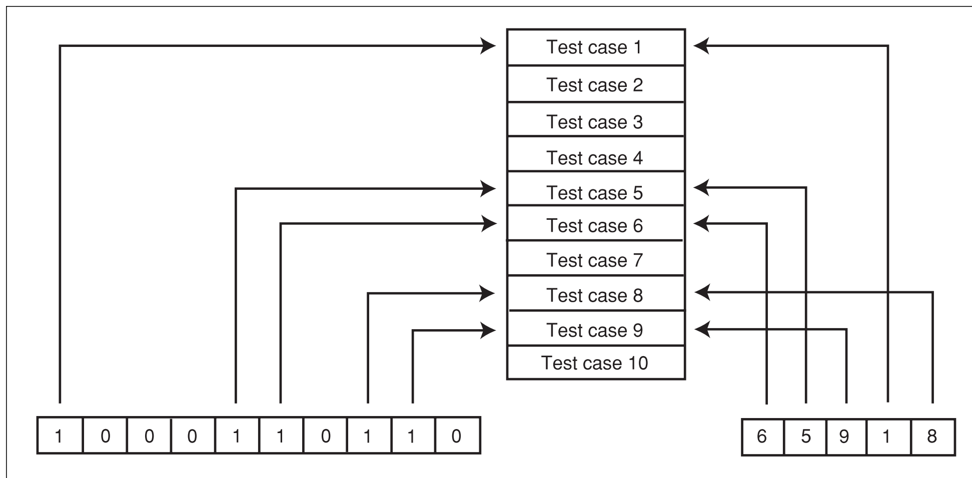


Fig. 2. Bit-string and pointer-set individuals

Test suite cost: test suite cost represents the execution cost of a test suite, which can mean execution time as well as hardware requirements.

Let T = {t1,t2,...,tn} be the test suite containing test cases t1,t2,...,tn , and R = {r1,r2,...,rk} the test requirements covered by this test suite. We assign the c : T → R positive function to each set of test cases. The execution cost of any given T set of test cases is then defined as

$$c(T)=\sum_{t\in T} c(t) \quad (1)$$

The cost of the individual test cases can be arbitrarily assigned, or measured during the mutation analysis phase. We will suppose here that the checking of each test requirement needs a certain amount of resources and execution time, and the initialization of each test case requires a certain amount of resources as well.

Thus the cost of a test case will be given by

$$c(t) = c1 + c2*L \quad (2)$$

where *c1* is the initialization cost, *c2* the cost required to check each test requirement, and *L* the number of covered test requirements.

Objective function: the function that evaluates the quality of each individual, and which the algorithm tries to minimize. To obtain the desired test suites, the objective function should take into account the following:

– Execution cost of the test suite should be minimized, by minimizing the redundancy in the test criteria covered by the test cases;
– The test suite should cover all test criteria.

Our objective function is the sum of the execution cost of all the test cases in the test suite, and a penalty for each untested requirement

$$O = c3*C + c4*M \quad (3)$$

where *C* is the cost of the individual, *M* is the number untested requirements, and *c3* and *c4* are weighting constants, which must be chosen so that it isn't economical to omit test cases.

2.2. Genetic Algorithm

The Genetic Algorithm is an optimization method trying to model the process of natural selection [7].

The canonical GA, which we employed here, works as follows:

```
Initialization
Create initial population
Evaluate of initial population
generation := 0

Generational loop
{
    Calculate fitness values
    Selection
    Recombination
    Mutation
    Evaluate of new individuals
    Insert new individuals into population

    generation := generation + 1
    } while generation < max. generation
```

The individuals are bit-string individuals, as the implementation of the crossover step was much more intuitive in this way.

Let's examine each algorithm step in detail:

Fitness: Fitness of individuals is evaluated by the linear rank-based method, where the *Fi* fitness of the ith individual is given by

$$F_i = 2 - sp + 2*(sp-1)*\frac{pos(fi)-1}{N_{ind}-1} \quad (4)$$

where *sp* is the selection pressure (here *sp*=*2*), *pos(fi)* is the position of the ith individual based on the value of the objective function, and N_{ind} is the population size.

Selection: Individuals are selected for breeding by the Stochastic Universal Sampling method: individuals are mapped on an axis where each individual has a length equal to its fitness.

We then generate a random number in the [1..nb_parents] interval, where *nb_parents* is the number of individuals we want to select for breeding. We then add i*(sum of all fitnesses)/(nb_parents) to this value, where i ∈ [0 .. nb_parents – 1], and select each time the individual to which this value points on the axis.

Recombination: We use the uniform recombination method: we generate a random bit pattern; bits of both parents are then inverted where the value of this mask is 1, giving the offspring.

Mutation: All offspring are mutated with a small probability to allow drastic changes. Beginning at a random position, we invert each bit of a predefined length section with *Pm* probability.

2.3. Pseudo-Bacterial Genetic Algorithm

Bacterial algorithms, developed in the second half of the '90s, model evolutional processes of bacteria. The simplest bacterial algorithm is the Pseudo-Bacterial Genetic Algorithm [8].

At the beginning of the algorithm, we create a random individual, on which we apply the bacterial mutation. We make $n-1$ copies (clones) of the original individual. Then we randomly select a part of the chromosome, which we mutate in each clone, but leave unchanged in the original individual. After the mutation, we evaluate each individual, and transfer the mutated part of the best individual to the other clones.

We repeat this mutation-evaluation-selection-reinsertion cycle until we have mutated all parts of the chromosome. We then select the best individual, and annihilate the others. We can repeat the cycle until we have a satisfactory solution, or we have reached a predefined generation number.

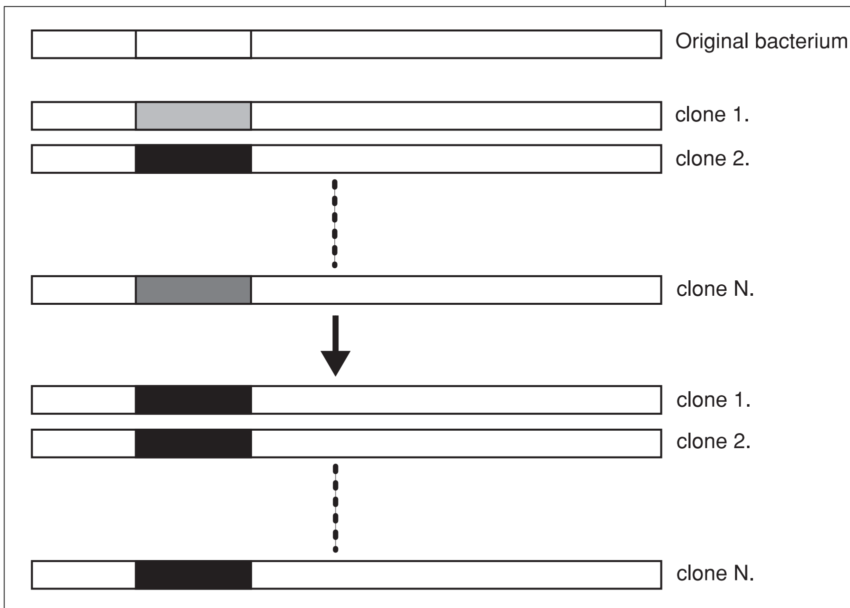


Fig. 3. The Pseudo-Bacterial Genetic Algorithm

We have implemented this algorithm with both types of individuals. In the case of bit-string individuals, the mutation step is the same as in the case of the GA. In the case of pointer-set individuals, a mutation has to allow changes in the length of the mutated individual (since individuals have no predefined length, and we have no a priori knowledge of the optimal length).

Thus, mutation can induce three kinds of modification:
- The substitution of a test case by another test case;
- The deletion of a test case;
- The addition of a test case.

2.4. Bacterial Evolutionary Algorithm

The Bacterial Evolutionary Algorithm is an improved version of the PBGA, which works on many individuals in parallel. It was inspired by the gene transfer ability of bacterial populations [9].

The algorithm works in the following way:
1. We create a random population of *n* individuals.
2. We apply the bacterial mutation (as shown in 2.3) on all individuals.
3. We apply the gene transfer operation *Ninf* times, where *Ninf* is the number of infections: during this step, we divide the population into an upper half (better individuals), and a lower half (worse individuals), and transfer genes from the upper half into the lower half.
4. We repeat steps 2-4. until we get a satisfactory solution, or reach the maximum number of generations.

In the case of this algorithm, we had to modify individuals so that they contain distinct genes, since the gene transfer operation requires a metric measuring how "good" each gene constituting the individual is. We took pointer-set individuals, and divided them into a predefined number of genes, which are groups of a variable number of test cases. We have implemented two different versions of gene fitness:

First version

In this first implementation, the goodness of a gene is given by the average cost at which a gene covers the requirements:

the fitness of a gene is given by

$$F=\frac{\sum_{i\in I} C_i}{R} \qquad (5)$$

where F is the fitness of the gene, C_i the cost of the test cases, I the set of test cases of the gene, and R the number of requirements covered by the gene.

During the gene transfer operation, we take one of the superior half of the bacteria, and insert its best gene to replace the worst gene of one of the bacteria of the lower half:

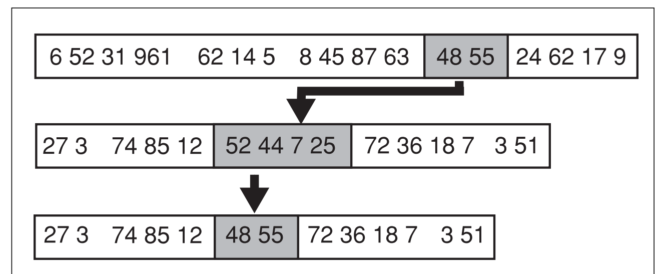


Fig. 4. Gene transfer 1

Second version

In this approach, we divide the test requirements in as many parts as there are genes in the bacteria. The goal is that each gene covers a specific part of the requirements. The goodness of a gene is defined in the same manner as the objective function of individu-

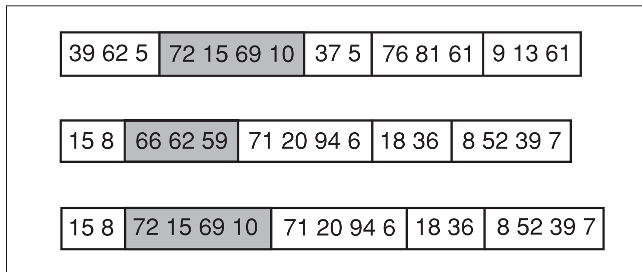
als in the previous cases, but the missed requirements are only taken into account in the interval covered by the gene. Thus, the fitness of a gene is given by

$$F = c1 * C + c2 * M_i \tag{6}$$

where F is the fitness of the gene, C is the cost of the gene, M_i is the number of missed requirements on the set to be covered by the gene, and $c1$, $c2$ are weighting constants.

During gene transfer, we take a bacterium from the upper half, and another from the lower half. We take a random gene of the source bacterium, and if it is better than the corresponding gene of the destination bacterium, we replace the corresponding gene of the destination bacterium with it:

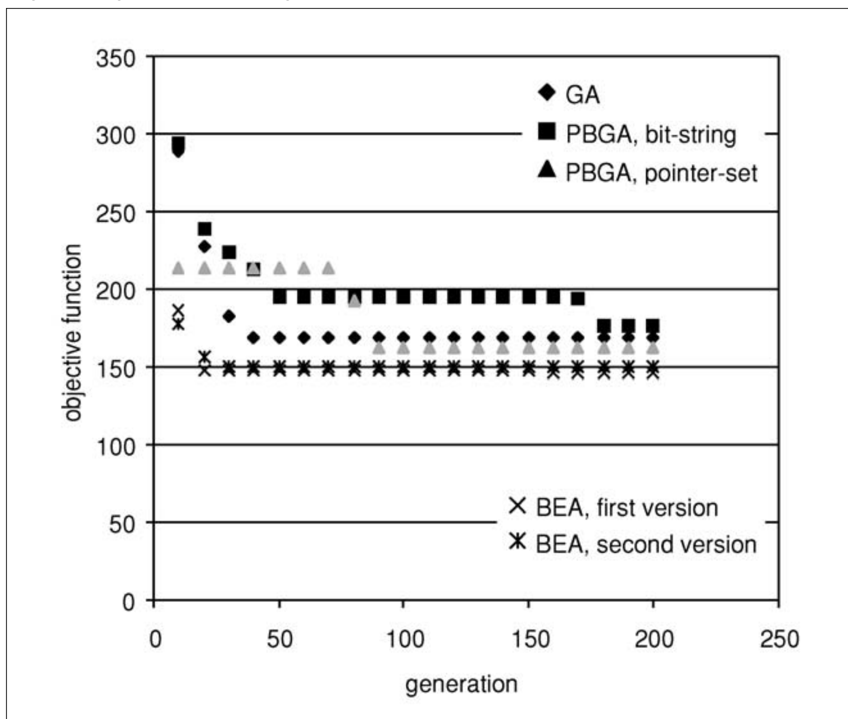
Fig. 5. Gene transfer 2



2.5. Algorithm Comparison

To compare the effectiveness of these algorithms in test case selection, we ran them on a fictive set of 100 test cases (as it will be shown later, the initial set of test cases for the INRES protocol only contains 41 test cases, which is too few to show differences in the convergence of these algorithms). The convergence of the different algorithms can be seen on *Fig. 6.:*

Fig. 6. Algorithm convergence



3. Automated test suite generation

We will now show the whole test generation process. We will illustrate this process by an example on the well-known INRES sample protocol.

The automated test generation process:

0) We create a formal specification of the protocol in SDL. Highly developed tools exist for this purpose ([3]). *Fig. 7.* shows the system overview of the INRES protocol SDL specification.
1) We apply a state-space exploration algorithm on the SDL specification, which gives us a highly redundant, unstructured set of MSC test cases.
2) With mutation analysis, we determine the matrix of test criteria for this set of test cases. *Fig. 8.* shows the full test suite resulting from the exploration of the SDL specification of the INRES system, containing 41 test cases, with the cost of individual test cases and the whole test suite, where the cost of test cases was calculated according to (2), with c1=20 and c2=5
3) We select an optimal subset of test cases from this set with one of the evolutionary algorithms presented above. This gives us a test suite that covers all test criteria, with minimal redundancy and execution cost. *Fig. 9.* shows the reduced set of test cases for the INRES protocol.

(Note: In this case, the selection of test cases is quite simple, and although it is not necessarily the case with very large test suites, all evolutionary algorithms found the same reduced test suite in a few generations.)

4. Conclusion

Conformance testing has become a crucial part in the development process of telecommunication protocols. Since the creation of test suites is a very time consuming process, automated test generation plays an increasingly important role in the development process.

We have shown here a complete method for the automatic generation of test suites from the SDL description of a system. We have only shown a simple example for illustration purposes. However, Mutation Analysis has been shown to work well on real-life cases ([4]) the motivational force behind the development of Evolutionary Algorithms was also the handling of extremely complex problems.

This test suite generation process is easily implementable, and should provide a working solution for the automated test generation of real-world telecommunication protocols.

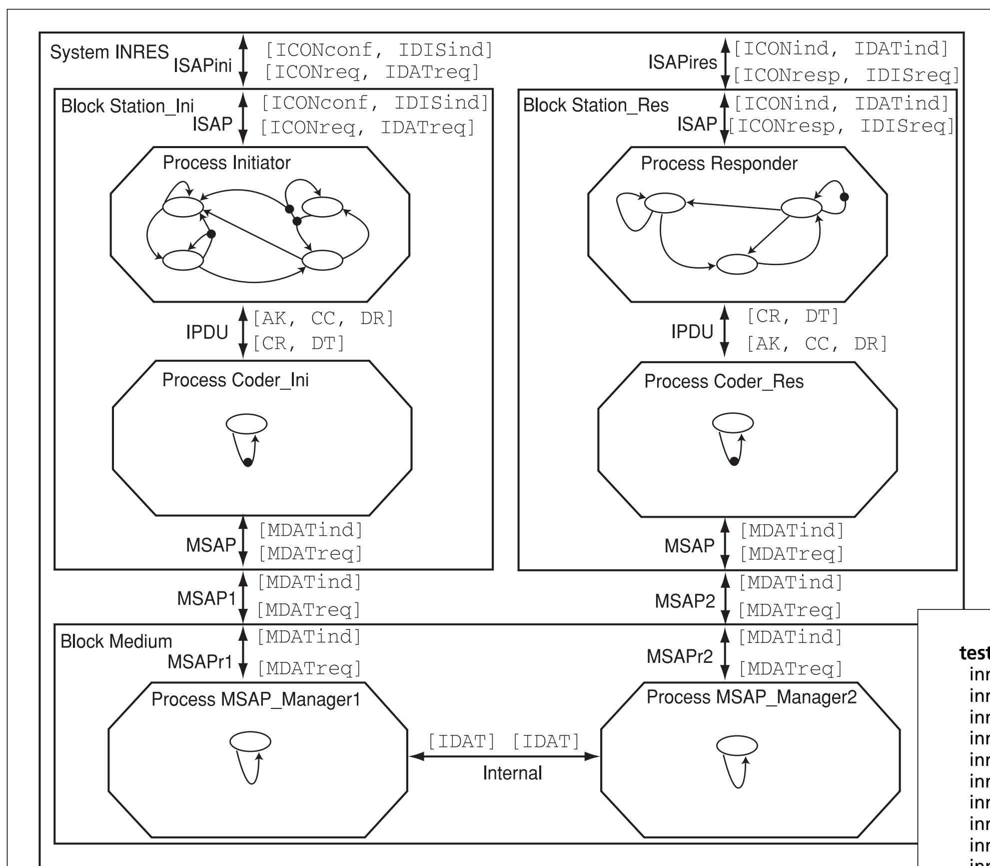


Fig. 7. Overview of INRES SDL specification

Fig. 8. Initial set of test cases

test case	cheked test criteria	test case cost
inres01	48	260
inres02	19	115
inres03	36	200
inres04	21	125
inres05	44	240
inres06	34	190
inres07	46	250
inres08	21	125
inres09	27	155
inres10	60	320
inres11	11	75
inres12	46	250
inres13	89	465
inres14	59	315
inres15	58	310
inres16	49	265
inres17	17	105
inres18	46	250
inres19	47	255
inres20	66	350
inres21	21	125
inres22	65	345
inres23	24	140
inres24	82	430
inres25	25	145
inres26	26	150
inres27	78	410
inres28	29	165
inres29	71	375
inres30	30	170
inres31	36	200
inres32	34	190
inres33	66	350
inres34	62	330
inres35	35	195
inres36	88	460
inres37	37	205
inres38	39	215
inres39	84	440
inres40	41	225
inres41	48	260
total test suite cost:		**10145**

Fig. 9. Reduced set of test cases

test case	cheked test criteria	test case cost
inres10	60	320
inres13	89	465
inres14	59	315
inres23	24	140
inres27	78	410
inres28	29	165
total test suite cost:		**1815**

References

[1] ITU-T. Recommendation Z.100:
Specification and Description Language (SDL), 1992.

[2] CCITT. Recommendation X.292:
The Tree and Tabular Combined Notation (TTCN), 1992.

[3] Telelogic Tau. http://www.telelocig.com

[4] Black P.E., Okun V., Yesha Y.:
Mutation Operators for Specifications.
In The Fifteenth IEEE International Conference on Automated Software Engineering 2000,
Proceedings ASE 2000, pp.81–88.

[5] Gábor Kovács, Zoltán Pap, Gyula Csopaki:
Automatic Test Selection based on CEFSM, 2002.
Acta Cybernetica 15, pp.583–599.

[6] B. Kotnyek, T. Csöndes:
Heuristic methods for conformance test selection.

[7] J.H.Holland:
Adaptation in Nature and Artificial Systems:
An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence,
MIT Press, Cambridge, 1992.

[8] M.Salmeri, M.Re, E. Petrongari, G.C.Cardarilli:
A Novel Bacterial Algorithm to Extract the Rule Base from a Training Set,
Dept. of Electronic Engineering, University of Rome, 1999.

[9] N.E.Nawa, T.Furuhashi:
Fuzzy System Parameters Discovery by Bacterial Evolutionary Algorithm,
IEEE Tr. Fuzzy Systems 7 (1999), pp.608–616.