Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms

André Baresel[1], Hartmut Pohlheim[1], and Sadegh Sadeghipour[2]

[1] DaimlerChrysler AG, Research and Technology, Methods and Tools
Alt-Moabit 96a, 10559 Berlin, Germany
{andre.baresel,hartmut.pohlheim}@daimlerchrysler.com
[2] IT-Power Consultants, Jasmunder Str. 9,
13355 Berlin, Germany
sadegh@itpower.de

Abstract. Evolutionary Testing (ET) has been shown to be very successful for testing real world applications [10]. The original ET approach focuses on searching for a high coverage of the test object by generating separate inputs for single function calls.
We have identified a large set of real world application for which this approach does not perform well because only sequential calls of the tested function can reach a high structural coverage (white box test) or can check functional behavior (black box tests). Especially, control software which is responsible for controlling and constraining a system cannot be tested successfully with ET. Such software is characterized by storing internal data during a sequence of calls.
In this paper we present the Evolutionary Sequence Testing approach for white box and black box tests. For automatic sequence testing, a fitness function for the application of ET will be introduced, which allows the optimization of input sequences that reach a high coverage of the software under test. The authors also present a new compact description for the generation of real-world input sequences for functional testing. A set of objective functions to evaluate the test output of systems under test have been developed. These approaches are currently used for the structural and safety testing of car control systems.

1 Introduction to Sequence Testing

When analyzing the code of real world applications, a large set of implementations can be identified which use a functional model whereby a controller procedure is called periodically. In the world of car control units it is very common for a software function to be based on an initialization and step function. Whereas the initialization function is only called once at the beginning, the step function is executed at regular time intervals e.g. every 10 milliseconds. A cruise control and vehicle-to-vehicle distance controller program (Distronic) which checks the velocity and distance to a leading vehicle at regular intervals to find out if a safe distance is maintained is an example of control software (structure shown in figure 7).

This kind of software seems to be very specific. However, an equivalent situation can be found within object oriented software. OO software systems implement objects which often create their own internal storage and methods. The methods initialize and alter the objects data during its life-time. It is very common for an object to provide an interface to initialize and change its data. The implemented functions behave differently on the basis of the object's inputs and internal settings.

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 2428–2441, 2003.

Depending on the complexity of the system under test automatic testing can be performed in three different ways. The approach commonly used is to generate pairs of initial states and input situations shown in Figure 1. This approach works well for simple software models but raises several problems. At first, the direct setting of internal states is not possible in all cases and needs changes to be made to program under test. By generating the internal states the test system has to make sure that the states produced are valid according to the specification. Forcing the system into a generated state is, in most cases, not useful for the tester. This is because the tester has to use the test data generated to analyze software bugs. He needs first to ascertain how to produce the initial state which causes the problem. This information is not provided by this test automation.

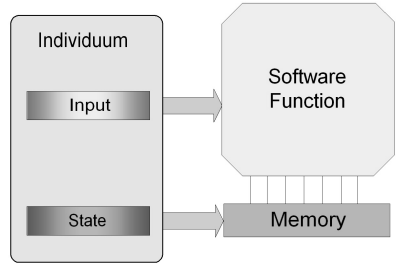

Fig. 1. Automatic testing by generating (state, input) pairs (first approach)

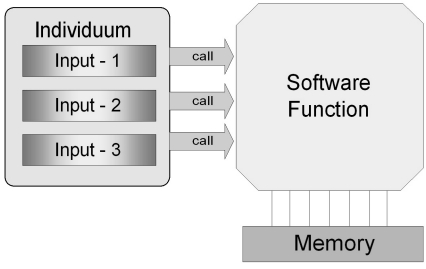

Fig. 2. Automatic testing by generating list of inputs (second approach)

For complex systems it is necessary to search for an input sequence. One approach is to create lists of inputs for sequential calls of the function under test. This is often sufficient for software systems based on state models, see Fig. 2.

For control systems requiring long 'real world' input sequences, encoded input functions must be generated. Figure 3 presents this third approach.

Monitoring the sequence test of a system differs from the original ET approach. For white box tests, a list of execution paths has to be analyzed. Black box tests are performed on sequences of output values instead of single values. This allows the assessment of dynamic functional and safety criteria.

The two approaches for generating input sequences are, from the tester's point of view, the best solution since they guarantee that the system is tested in the same way as it will later be used.

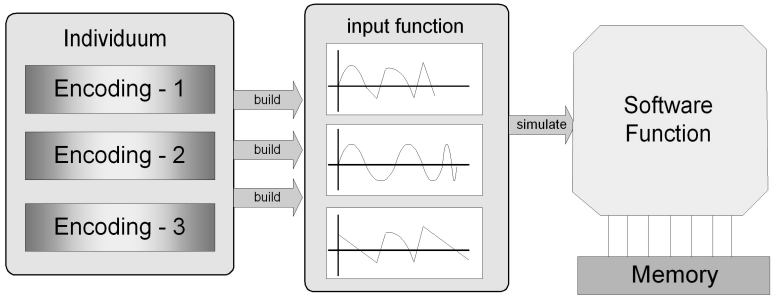

Fig. 3. Automatic testing by generating input sequences from encoded input functions (third approach)

Section 2 describes how evolutionary algorithms (EA) have been applied to structural and functional sequence testing. Both applications have been implemented by the authors and results of real world experiments will be presented. Structural Sequence Testing has been tested with the generation of input sequences, see section 3. Functional Sequence Testing, see section 4, has been applied to Safety Tests by creating encoded input functions.

2 Overview of Evolutionary Testing (ET)

Evolutionary algorithms (EA) have been used to search for data for a wide range of applications. EA is an iterative search procedure using different operators to copy the behavior of the biologic evolution. Using EA for a search problem it is necessary to define the search space and the objective function (fitness). The algorithms are implemented in a widely used tool box [7]. It consists of a large set of operators e.g. real and integer parameter, migration and competition strategies.

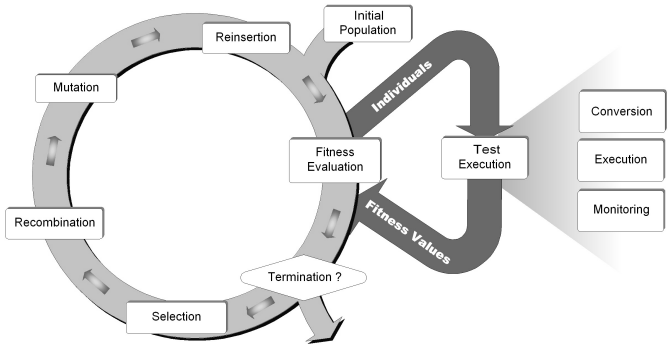

Fig. 4. The structure of Evolutionary Testing

Evolutionary Testing uses EA to test software automatically. The different software test criteria formulate requirements for a test case set to be generated. Until now, the generation of such a set of test data usually had to be carried out manually. Automatic software testing generates a test data set automatically, aiming to fulfill the requirements in order to increase efficiency and resulting in an enormous cost reduction.

Evolutionary Structural Testing. Structural Testing has the goal of automating the test case design for white box testing criteria [2]. Taking a test object, namely the software under test, the goal is to find a test case set (selection of inputs) which achieves full structural coverage. The general idea is a separation of the test into test aims and the use of EA to search for test data fulfilling the test aims.

The separation of the test into partial aims and the definition of fitness functions for partial aims are performed in the same manner for each test criterion. Each partial aim represents a program structure that requires execution in order to achieve full coverage of the structural test criterion selected, i.e. each single statement represents a partial aim when using statement coverage.

The definition of a fitness function, that represents the test aim accurately and supports the guidance of the search, is a prerequisite for the successful application of Evolutionary

Test. In order to define the fitness function, this research builds upon previous work dealing with branching conditions (among others [8], [5], and [9]). These are developed in [10] by introducing the idea of an approximation level.

Evolutionary Functional Testing. Complex dynamic systems must be evaluated over a long time period (longer than the highest internal dead time or time constant). This means that the systems must not be stimulated for only few simulation steps, but rather the input signals must be up to hundreds of time steps long. Long input sequences are therefore necessary in order to simulate these systems.

Several disadvantages result from the length of the sequences necessary: the number of free variables during optimization is very high and the correlation between the variables is great. For this reason, one of the most important aims is the development of a very compact description for the input sequences. This must contain as few elements as possible but, at the same time, offer a sufficient amount of variety to stimulate the system under test as much as necessary.

Moreover, possibilities for automatically evaluating the system answers must be developed, which allow differentiation between the quality of the individual input sequences. These requirements and the solutions developed will be presented in section 4.

3 Evolutionary Algorithms for Structural Sequence Testing

The evolutionary structural test was designed originally to find a set of test data for single function calls which results in a high structural coverage of the software under test. This approach has been defined for the most common test criteria in [10]. The fundamental idea is the transformation of the test data generation into a search problem which is then solved by evolutionary algorithms. The optimization function designed for this calculates so-called fitness values. The evolutionary algorithm uses these values to optimize the solution to meet the test aim currently selected.

Structural Sequence Testing targets the structural coverage of a given test object. For this reason, test aims are defined in the same manner as introduced for ET. However, in order to apply evolutionary algorithms it is necessary to redefine the search space and the fitness function because test cases are now input sequences each executing several paths (see figure 5).

3.1 Search Space Structure

In the original approach the search space is formed by the input parameters of the function under test. This is different for sequence tests. The authors decided to implement the generation of input sequences for the first experiments in the area of structural tests. With this approach one test datum is a list of inputs. Monitoring a test case will return a list of execution paths (one per input). The length of this list is not defined in the test object and is, in general, not limited for control systems.

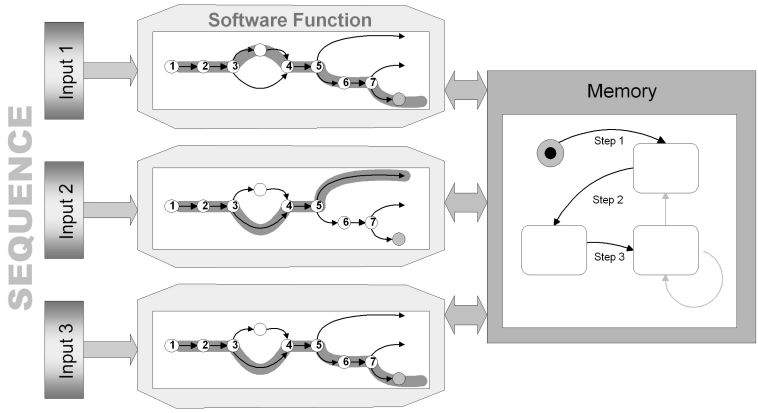

Fig. 5. Test case (sequence) and different execution paths for the inputs; On the right hand side an example memory model is shown, where each step results in a state change

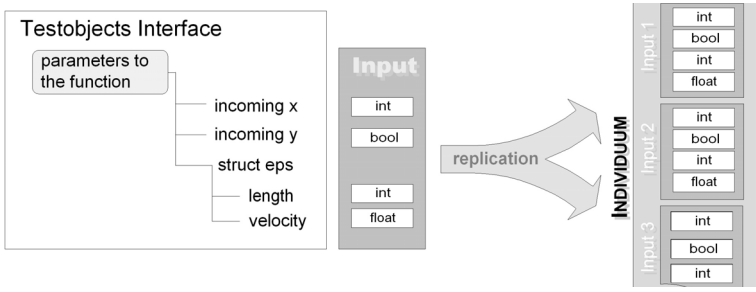

Fig. 6. The encoding of the test case data to different runs

For evolutionary algorithms, an encoding of the individuals generated (*mapped to test data*) has to be defined. A straightforward approach is to let the user specify the length of the input sequences manually. With the information on the sequence length a search space is formed by replicating all input parameters in such a way that a separate value will be provided for each call and each parameter. Every individual generated by the EA represents the data of one sequence of calls (here called test data). Figure 6 shows the mapping of the data.

3.2 Sequence Evaluation

As mentioned previously, the test criteria for structural coverage are defined on the basis of the test object source code. An automatic test system has to monitor and analyze all runs of the test object to find out which test aims have been reached (structures that have been covered) and to provide a fitness value for the test aim currently selected. This fitness is determined by monitoring the test execution.

In the original ET approach, fitness is calculated by analyzing one execution path through the test object. This is different in sequence testing, since each test case run results in a list of execution paths. This list is the basis upon which a fitness value has to be calculated. The sequence testing fitness function introduced here uses the original idea of the fitness function and creates a sequence fitness value from that.

For structural coverage it is not important at which step of a sequence a structural element is covered. For this reason, the authors decided to design a fitness function which analyzes all the execution paths of a test case sequence and uses the closest path to the test aim as the fitness of the sequence. With this approach it does not matter whether a close execution path is traversed earlier or later in the sequence and whether or not the EA is able to create better performing solutions by ordering the sequence in different ways (see [1] for details of sequence optimization).

The original ET approach uses decisive branches to ascertain the fitness of an execution path (see [10]). This idea can be reused by analyzing each path of the sequence and determining an overall fitness. Due to the fitness function's structure, it has to be minimized by the EA. For this reason, the overall fitness is the minimum of the path fitness values. Using this approach the best path will define the fitness of the sequence.

3.3 Experiments

The experiments on Structural Sequence Testing were performed with an extended version of the automatic structural test system. The GEATbx [7] is the optimization component used by the system. The authors applied the standard settings for this class of problem (namely, 6 subpopulations with 50 individuals each, linear ranking, a selection pressure of 1.7, migration and competition between subpopulations). A real valued representation is used. The subpopulations employ different search strategies by using different settings for the recombination (discrete and line recombination) and mutation operator (real valued mutation with differently sized mutation steps – large, medium and small).

Table 1. Information on the complexity of the test objects

Name	Size	LOC	nodes	param.	if–then	conditions	nesting level
Ctrl_S	16kB	220	20	28	5	6	2
Enable_S	54kB	800	140	15	51	70	10
Enable_A	44kB	520	86	11	39	56	8

Three software functions which are part of a large real-world functional model were tested using the extended ET system. Table 1 provides an overview of the complexity measures for the test objects. The test object *Ctrl_S* is relatively small but is code generated from a state diagram containing flags in all conditions.

Table 2. Experimetal results, showing the numbers of test aims, number of individuals generated and reached coverage

	Coverage criteria	test aims	num. individuals	coverage	not reached
Ctrl_S	StatementCover	17	550 000	95 %	1
	BranchCover	23	650 000	91 %	2
	ConditionCover	12	370 000	92 %	1
Enable_S	StatementCover	135	225 000	99 %	1
	BranchCover	196	550 000	98 %	5
	ConditionCover	168	650 000	97 %	5
Enable_A	StatementCover	87	390 000	95 %	5
	BranchCover	126	495 000	92 %	11
	ConditionCover	116	425 000	94 %	8

The three standard test criteria were employed. Statement cover requires a test case set that executes all statements of the software under test. The branch cover criterion requires

a test case set traversing all branches of a test object. Last but not least, condition cover necessitates a collection of test data evaluating all the atomic conditions of a program with the values *true* and *false*.

The high coverage reached for all the criteria can be seen in table 2. Only a few of the test aims not reached are a sequence test specific problem. Some of them refer to the problem of unreachable code created by the code generator and some to the problem of performing an evolutionary test with flag conditions. However, in the next paragraphs we would like to give an short overview of the reasons why certain test goals were not reached.

Test result details for test object 'Ctrl_S'. This test object was difficult to test with the original ET approach due to the source code structure. The code was generated from a state diagram. All the program's conditional statements use flags that have been assigned previously. Some conditions access flags that have been assigned in previous calls of the function which result in the test object requiring sequence tests. The test runs have shown that this kind of software does not pose a problem for the approach, since all sequence test specific test aims have been covered. For one statement, the corresponding two branches and the flag condition of a sequence could not be found because the search was not guided to a flag assignment within a function call.

Test results for test object 'Enable_S'. The testing of this module demonstrates the potential of the evolutionary testing approach. The test object consists of 800 lines of code leading to 135 control flow nodes, 196 branches and 168 test aims for simple condition cover.

During the statement coverage test only one statement was not executed. The non-executable statements, branches or conditions appear when using current versions of code generators. This occurs if the developer uses template library functions and configures them with constants. This leads to a comprehensible model with reuse of components but the code generated contains lines of code that are not executable.

The performance of the branch coverage test was more challenging, resulting in coverage of 98%. Only five branches were not traversed. Upon checking the code we found that two branches belong to the non-executable statement. One branch is not executable because the associated condition cannot be evaluated as *false*. The evolutionary test system was not able to find an input for two conditions which depend on sequential calls. This is due to the high nesting level of dependent variable assignments and uses. At the moment it is not possible to guide the search to a solution that executes an assignment in one sequence step and traverses the corresponding condition in later steps. This would require further research.

The results of simple condition cover were good in that the evolutionary test found a test case set covering 97% of the test aims. For the reason why the five conditions not covered were missed, see the paragraph on branch coverage above.

Test results for test object 'Enable_A'. This module is not particularly complex with regard to metrics, however, it contains some program structures that are difficult to test. First of all, the code is state oriented and many conditions depend on the settings of state variables. The function has a high nesting level of *if-then-else* statements and employs a state encoded using a set of flags. Again, the test object contained unfeasible code because of the use of library templates (3 statements). Two other statements were not covered because of the flags used in the conditions. The results of branch coverage are similar to those of statement coverage. The test case set found did not cover 4 branches start-

ing at flag conditions and 6 branches are placed at unreachable code. One branch could not be traversed because a precondition had not been satisfied. This branch requires an input sequence assigning two variables in previous calls of the function. The approach does not find a solution for this. The condition cover test returned equivalent results. Four of the conditions are not feasible because of the generation of code. The test system did not find an input sequence covering three flag conditions. Again, the condition which requires variable settings in previous calls was not fulfilled.

4 Evolutionary Algorithms for Functional Sequence Testing

The generation of test sequences for dynamic real-world systems pose a number of challenges:

- The input sequences must be long enough to stimulate the system realistically.
- The input sequences must possess the right qualities in order to stimulate the system adequately. This concerns aspects such as the type of signal, speed and rate of signal change.
- The output sequences must be evaluated regarding a number of different conditions. These conditions are often contradictory and several of them can be active simultaneously.

In order to develop a test environment for the functional test of dynamic systems, which can be applied in practice, the following main tasks must therefore be completed.

- Creation of a compact description for the input sequences,
- Evaluation of output sequences regarding defined requirements,
- Inclusion of input sequence generation and evaluation of system requirements into an optimization process.

Throughout the whole section examples are given using a real-world system. The structure of the integrated cruise control and vehicle-to-vehicle distance control (Distronic) is shown in figure 7. An extensive description is contained in [3].

The test environment presented in this section is implemented in Matlab and used for the testing of Simulink/Stateflow models. However, we concentrate on the presentation of the main concepts and the demonstration of results.

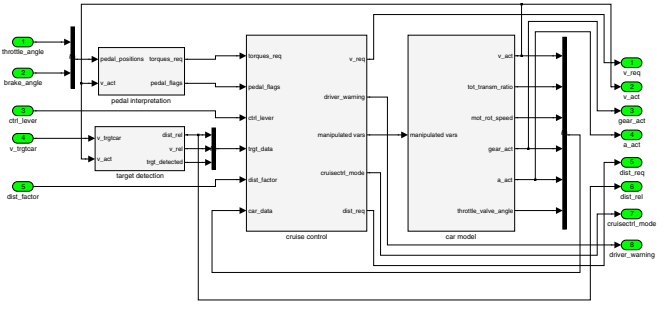

Fig. 7. Model of an integrated cruise control and vehicle-to-vehicle distance control (Distronic)

4.1 Description of Input Sequences

In order to describe the input sequences, several descriptions must be differentiated. On the one hand we have the description of the signal, which is used as input for the simulation of the dynamic system (simulation sequence). On the other hand we have the compact description which the user stipulates (compact user description). Between these is the description of the boundaries of the variables for the optimization (description of sequence bounds) as well as the instantiation of the individual sequences (optimization sequence). These different sequence description levels are illustrated in figure 8.

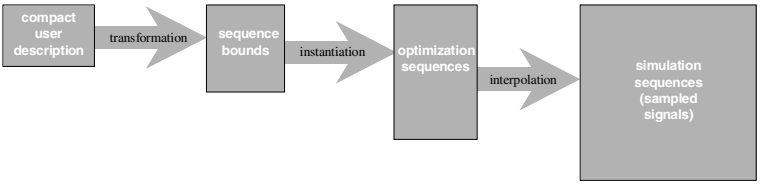

Fig. 8. Different levels of input sequence description

For a compact description the long simulation sequence is subdivided into individual sections. Each section is represented by a base signal, which is parameterized by the variables: signal type, amplitude and length of section. We use the following base signal types: step, ramp (linear), impulse, sine, spline. These base signal types are sufficient to generate any signal curve.

Only the possible areas for these parameters are specified for the optimization. In this way, the bounds are defined, within which the optimization generates solutions, which are subsequently evaluated by the objective function with regard to their quality.

The amplitude of each section can be defined absolutely or relatively. The length of the section is always defined relatively (to ensure the monotony of the signal). The base types are given as an enumeration of possible types. These boundaries must be defined for each input of the system. For nearly every real-world system these boundaries can be derived from the specification of the system under test.

An example of an input sequence description is provided in figure 10. The textual description as defined by the tester is given. Figure 10 provides examples of three different input signals generated by the optimization.

```
Input.Names = {'throttle_angle', 'brake_angle', 'ctrl_lever', ...
               'v_trgtcar', 'dist_factor'};
Input.BasePoints = [10, 10, 10, 10, 2]
Input.Amplitude.Bounds = [  0,   0, 0,  0, 1;...
                          100, 100, 3, 30, 1]
Input.Amplitude.Interpret = {'abs', 'abs', 'abs', 'abs', 'abs'};
Input.Basis.Bounds = [1, 1, 1, 1, 1; ...
                      3; 3; 5; 3; 3]
Input.Transition.Pool = { {'linear', 'spline'}; {'linear'}; {'impulse'}; ...
                          {'spline'}; {'step'} };
Sim.Length = 200; Sim.SamplingRate = 2.5;
```

Fig. 9. Textual description of input sequences

The system under test (Distronic) has 5 inputs. We use 10 sections for each sequence. The amplitude for the throttle pedal can change between 0 and 100, the control lever can only have the values 0, 1, 2 or 3.

In real-world applications the variety of an input signal is often constrained with regard to possible base signal types. An example of this is the control lever in figure 9. This input contains only impulses as the base signal type.

To generate a further bounded description it is possible to define identical lower and upper bounds for some of the other parameters. In this case, the optimization has an empty search space for this variable – this input parameter is a constant. An example is the distance factor (a measure for the relative distance to the preceding vehicle) in figure 10. This input signal is always set to a constant value of 1. Thus, it is not part of the optimization (but used as a constant value for the generation of the internal simulation sequence).

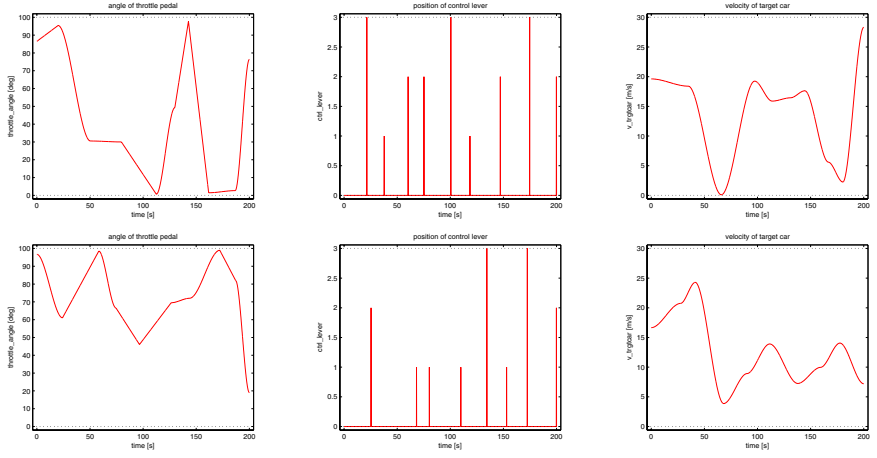

Fig. 10. Instances of simulation sequences generated by the optimization based an the textual description in figure 9; left: throttle pedal, middle: control lever, right: velocity of target car

All these different levels of the description of the input sequences ensure that the requirements for an adequate simulation of the system and a very compact and comprehensible description by the tester are fulfilled. The compact description is used for the optimization ensuring a small number of variables. When comparing the size of both descriptions for the example dynamic system used (5 inputs – one of the inputs is constant, 10 signal sections, 3 variable parameters for each section, 200 seconds simulation time, sampling rate 2.5 Hz) the differences are enormous:

$$\begin{aligned} SizeSimulationSignal &= 5 \cdot 200 \cdot 2.5 = 2500 \\ SizeCompactDescription &= (5-1) \cdot 10 \cdot 3 = 120 \end{aligned} \qquad CompressionRatio = \frac{2500}{120} = 20.8 \tag{1}$$

Only this compact description opens up the opportunity to optimize and test real-world dynamic systems within a realistic time frame.

4.2 Evaluation of Output Sequences and Objective Function

The test environment for the functional test of dynamic systems must perform an evaluation of the output sequences generated by the simulation of the dynamic system. These output sequences must be evaluated regarding the optimization aims. During the test we always search for violations of the defined requirements. Possible aims of the test are to check for violations of:

- signal amplitude boundaries,
- signal dependencies,
- maximal overshoot and maximal settlement time.

Each of these checks must be evaluated over the whole or a part of the signal lengths and an objective value generated. Due to space constraints we describe only the first two requirements in detail. However, our test environment can assess all of the checks (and more will be added).

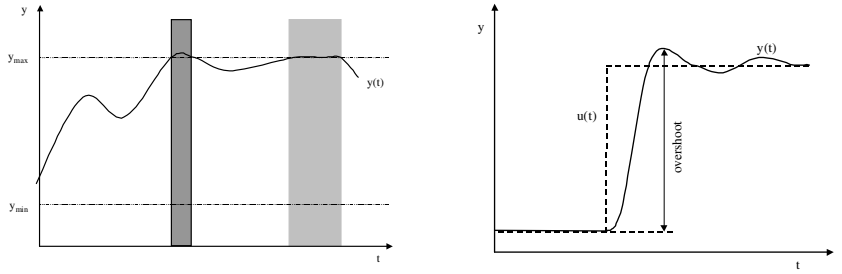

Fig. 11. Left: violation of maximum amplitude; right: assessment of signal overshoot

An example of the violation of signal amplitude boundaries is given in figure 11, left. A minimal and maximal amplitude value is defined for the output signal y. In this example the output signal violates the upper boundary. The first violation is a serious violation as the signal transgresses the bound by more than a critical value y_c (parameter for this requirement). In this case, a special value indicating a severe violation is returned as the objective value (value -1), see equation (2). The second violation is less severe, as the extension over the bound is not critical. At this point an objective value indicating the closeness of the maximal value to the defined boundary is calculated. This two-level concept allows a differentiation in quality between multiple output signals violating the bounds defined. The direct calculation of the objective value is provided in equation (2).

$$signal_{\max} = \max(y(t)) \qquad ObjVal_{\max} = \begin{cases} -1 & signal_{\max} \geq y_{\max} + y_c \\ \left(\dfrac{signal_{\max}}{y_{\max}} \right)^6 & signal_{\max} < y_{\max} + y_c \end{cases} \tag{2}$$

A similar assessment is used for calculating the objective value of the overshoot of an output signal after a jump in the respective reference signal. First, the maximal overshoot value is calculated. Next, the relative height of the overshoot is assessed. A severe overshoot outside the specification returns a special value (again –1). This special value is used to terminate the current optimization. The test was successful, as we were able to find a violation of the specification and thus reach the aim of the optimization. In all other cases, an objective value equivalent to the value of the overshoot is calculated (similar to equation (2)).

Each of the requirements tested produces one objective value. For nearly every realistic system test we receive multiple objective values. In order to assess the quality of all objectives tested we employ multi-objective ranking as supported by the GEATbx [7]. This includes Pareto-ranking, goal attainment, fitness sharing and an archive of previously found solutions.

4.3 Experiments

The test environment was used for the functional testing of a number of real-world systems. One of these is the Distronic model described earlier ([3], structure shown in figure 7), for which the results of one test will be presented in this subsection.

For the car system with an activated Distronic the maximum speed was specified at 44 m/s (the critical value y_c was set to 0). This means that a higher speed is not permitted under any circumstances. Thus, a test was specified to search for inputs which produce a speed greater than this boundary.

With an active Distronic the car can be accelerated only by pushing the control lever upwards (the respective input value is 1). The car is decelerated by pushing the control lever downwards (input value: 2). Beside the amplitude of the input control lever, the relative length of the signal sections could be changed between 1 and 5. The results of one successful optimization are shown in figures 12 and 13.

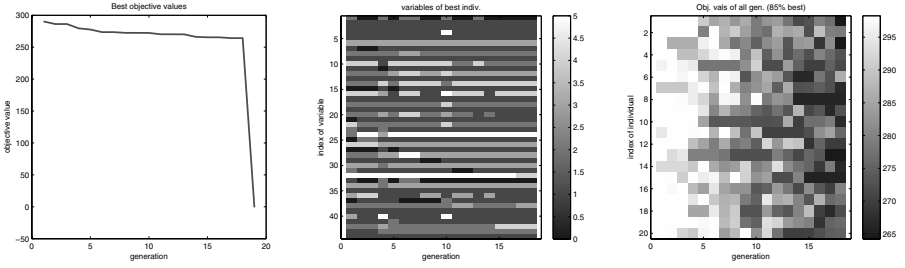

Fig. 12. Visualization of the optimization process, left: best objective value; middle: variables of the best individual; right: objective value of all individuals

The optimization process is visualized in figure 12. The left graphic presents the progress of the best objective value over all the generations. The optimization continually finds better values. In the 19th generation a serious violation is detected and an objective value of –1 returned. The middle graphic presents the variables of the best individual in a (gray) color quilt. Each line represents the value of one variable over the optimization. The graphic on the right visualizes the objective values of all the individuals during the optimization (generations 1 to 18).

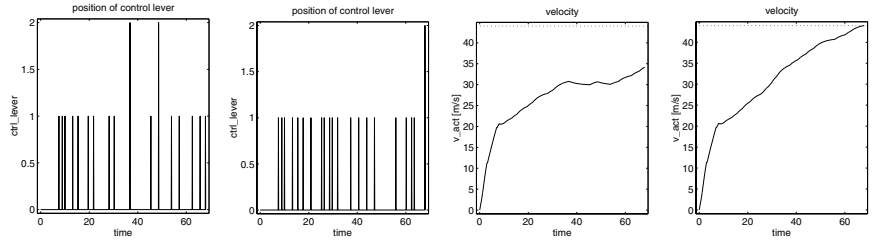

Fig. 13. Problem-specific visualization of the best individual during the optimization, left: input of the control lever - begin and end of optimization (2nd and 19th generation), right: vehicle velocity - begin and end of optimization (2nd and 19th generation);

The graphics in figure 13 provide a much better insight into the quality of the results, visualizing the problem-specific results of the best individual of each respective generation. The input of the control lever is shown in the two left side graphics. The resulting

velocity of the car is presented at the right. The graphics are taken from the 2nd (left) and 19th generation (right). At the beginning the maximal velocity is far below the critical boundary. During the optimization the velocity is increased (the control lever is pushed up more often and at an earlier stage as well as being pushed down less frequently). At the end an input situation is found, in which the velocity is higher than the bound specified. By looking at the respective input signals the developer can check and change the implementation of the system.

During optimization we employed the following evolutionary parameters: 20 individuals in 1 population, discrete recombination and real valued mutation with medium sized mutation steps, linear ranking with a selection pressure of 1.7 as well as a generation gap of 0.9. Other tests with a higher number of relevant input signals and thus more optimization variables employ 4-10 subpopulations with 20-50 individuals each. In this case, we use migration and competition between subpopulations. Each subpopulation uses a different strategy by employing different parameters (most of the time differently sized mutation steps).

5 Conclusion

In this paper we have presented different approaches for applying Evolutionary Testing to sequence testing. The first method aims at automating test case generation to achieve high structural coverage of the system under test (white box test). The other approach searches for input sequences violating the specified functional behavior of the system (black box test).

Both test methods were implemented. Experiments with software modules and dynamic systems with varying complexity were conducted. We have presented a small selection of the results. The results show the new test methods to be promising. It was possible to find test sequences, without the need for user interaction, for problems which could previously not be solved automatically.

During the experiments a number of issues were identified which could further improve the efficiency of the test methods presented. For the structural sequence test, a fitness function which expresses the dependency between successive sequence steps would significantly aid the search process. In the case of the functional sequence test, it is necessary to include as much as possible of the existing problem-specific knowledge into the optimization process.

References

[1] *Baresel, A., Sthamer, H., Schmidt, M.*: Fitness Function Design to improve Evolutionary Structural Testing. Proceedings of GECCO2002, New York, USA, pp. 1329–1336, 2002.

[2] *Beizer, B.*: Software Testing Techniques. New York: Van Nostrand Reinhold, 1983.

[3] *Conrad, M., Hötzer, D.*: Selective Integration of Formal Methods in the Development of Electronic Control Units. Proceedings of Second IEEE International Conference on Formal Engineering Methods ICFEM'98, IEEE Computer Society, pp. 144–155, 1998.

[4] *Harman, M., Hu, L., Munro, M., Zhang, X.*: Side-Effect Removal Transformation. IEEE International Workshop on Program Comprehension (IWPC) Toronto, Canada, 2001.

[5] *Jones, B.-F., Sthamer, H., Eyres, D.*: Automatic structural testing using genetic algorithms. Software Engineering Journal, vol. 11, no. 5, pp. 299–306, 1996.

[6] *Korel, B.*: Automated Test Data Generation. IEEE Transactions on Software Engineering, vol. 16 no. 8, pp. 870–879, 1990.
[7] *Pohlheim, H.*: *GEATbx* - Genetic and Evolutionary Algorithm Toolbox for *Matlab*. http://www.geatbx.com/, 1994-2003.
[8] *Sthamer, H.*: The Automatic Generation of Software Test Data Using Genetic Algorithms. *PhD* Thesis, University of *Glamorgan, Pontyprid*, Wales, Great Britain, 1996.
[9] *Tracey, N., Clark, J., Mander, K., McDermid, J.*: An Automated Framework for Structural Test-Data Generation. Proceedings of the 13th IEEE Conference on Automated SE, Hawaii, USA, 1998.
[10] *Wegener, J., Sthamer, H., Baresel, A.*: Evolutionary Test Environment for Automatic Structural Testing. Special Issue of Information and Software Technology, vol. 43, pp. 851–854, 2001.
[11] *Wegener, J., Sthamer, H., Jones, B., Eyres, D.*: Testing Real-time Systems using Genetic Algorithms. Software Quality Journal, vol. 6, no. 2, pp. 127–135, 1997.