An Event-Driven Test Framework for Evolutionary Algorithms in Dynamic Environments

A. Şima Etaner-Uyar
Computer Engineering Department
Istanbul Technical University
Maslak TR 34469 Istanbul - Turkey
Email: etaner@cs.itu.edu.tr

H. Turgut Uyar
Computer Engineering Department
Istanbul Technical University
Maslak TR 34469 Istanbul - Turkey
Email: uyar@cs.itu.edu.tr

Abstract— **There is a growing interest in applying evolutionary algorithms to dynamic environments. This can be seen by the increase in the number of papers addressing various issues concerning different types of changing environments. Most of these studies use their own test functions and implementations which makes it harder to repeat the reported experiments. This paper focuses on the types of benchmark functions found in literature that are used for testing evolutionary algorithms in dynamic environments and proposes an easy-to-use test framework. The proposed test framework consists of an event driven environment control mechanism component as well as a scenario creator component which allows the user to experiment with different instances of a problem, incorporating various properties and types of change. The components of the framework interact with each other through a well defined function call interface. By way of these features, the test framework realizes its design goals which are ease of use, compatibility, controllability, flexibility and generalizability. The test framework is still in its early stages of development. Extensions and modifications will be made based on the requirements and feedback received from its users.**

I. Introduction

There is a growing interest in applying evolutionary algorithms to dynamic environments. This can be seen by the increase in the number of papers addressing different issues concerning different types of changing environments. Most of these studies report results based on simple test functions devised specifically for that study. These results are commonly given to show the algorithms' ability and speed to adapt to the change without much concern for the type and dynamics of the change in the system. These issues make it hard to repeat the tests and experiments performed in these studies and thus prevent healthy comparisons with other approaches in future studies. Another problem is that since very simple test problems are designed for testing some specific properties of an approach, it is hard to make generalizations to real world problems.

Designing and using standard, tunable test problem generators has recently become an area of interest within the evolutionary computing community. In Genetic and Evolutionary Computation Conference held in 2003 (GECCO 2003), one of the main discussion topics of the "Evolutionary Algorithms for Dynamic Optimization Problems Workshop (evoDOP-2003)" [17] was the design and use of suitable benchmark problems for different types of changing environments. It can be seen from the discussions that emerged as a result of the workshop that the EA community is lacking in appropriate and easy to use benchmark test problems for different dynamic problem classes.

The main aim of this study is to determine the type of benchmark functions for changing environments as used by different researchers and propose a new test framework for testing different evolutionary approaches. The proposed test framework provides the basic features expected from such a system, is easy to use and can be applied to various discretely changing problems using different representations. It will also be possible to make more accurate comparisons between EA approaches by using the benchmark functions repository included in the framework.

The rest of the paper is organized as follows: Section 2 groups different types of problems usually encountered in dynamic optimization studies and categorizes them based on their specific requirements. Section 3 lists the properties expected from a good benchmark function and provides a survey on previously suggested benchmark functions and test problem generators. Section 4 explains the proposed test framework and discusses it in detail. Section 5 demonstrates how the framework can be used. Section 6 discusses the properties of the framework, showing how it realizes the desired properties of a good test framework for dynamic environments. Section 7 concludes the paper and provides directions for future releases.

II. Types of Dynamic Environments

Different types of dynamic environments have different requirements. These environments can be categorized in several ways. However, for the purposes of designing benchmark functions and test problem generators, classification can be done based on the type of change required. Two main groups can be used for this purpose:

- The first group contains the continuous environments which usually use floating point representations.
- The second group is composed of discrete problems which may also be termed as combinatorial optimization problems. In evoDOP-2003 workshop [17], suggestions were made to further divide this group into three subgroups as allocation problems, routing problems and sequencing problems.

0-7803-8515-2/04/$20.00 ©2004 IEEE

Common to all groups, change in the environment can be categorized based on the following criteria as given in [5]:

- frequency of change
- severity of change
- predictability of change
- cycle length / accuracy

When designing test problems and benchmark functions, the above groups and categorization criteria need to be taken into consideration.

III. Test Problem Generators and Benchmark Functions

Several benchmark functions and test problems have been used in literature for exploring dynamic environments and for comparing different evolutionary algorithm (EA) approaches. Most studies commonly use one of the following test problems for comparisons:

- For discrete environments, two of the most commonly used benchmark functions in literature are the dynamic bit matching problem (e.g. used in [6]) and the dynamic knapsack problem (e.g. used in [7], [10], [11], [12]).
- For continuous environments, some of the most commonly used benchmark problems are the moving parabola problem (e.g. used in [1]), the moving peaks benchmark function [3] (e.g. used in [4]) and DF1 [8] (e.g. used in [9]).

A detailed survey of test problems and benchmark functions used in literature can be found in [2] and [5].

In most studies that use specific test functions, one major concern is the fact that their implementations are also specific to the EA approach they are used in conjunction with. This increases the problem of not being able to accurately repeat reported experiments in future studies for comparison purposes. Based on these observations, tunable test problem and fitness landscape generators have been proposed by Branke [3], Morrison [8] and Smith [13], addressing several test problem generation issues. The properties expected from a good test problem generator have been studied in [2], [3], [8], [13]. Based on these studies and current experience of the authors, these properties can be listed as follows:

- allow controlled changes, i.e. the optimal value at each change instance should be known and the environment should be tunable using parameters
- allow different types of changes
  - changes of different severity levels
  - periodic changes
  - oscillating changes
  - random changes (based on a chosen distribution)
- allow extensions to real-world problems
- be easy to implement and to use
- be platform-independent
- be efficient

In the following sections, the proposed test framework will be evaluated based on these desirable properties.

IV. The Proposed Dynamic Environment Test Framework

The proposed test framework is designed to generate discrete dynamic problem instances, so it will henceforth be called DEFEAT (Dynamic Environment Framework for Evolutionary Algorithm Testing).[1] DEFEAT is actually more than just a test problem instance generator. It works according to an event-based environment control approach where each change instance is regarded as an event in the system and different event scenarios can be created.

The main considerations in the design of DEFEAT are ease of use, compatibility, controllability, flexibility and generalizability. To implement these considerations, DEFEAT is organized as a set of components which interact with each other through a well defined function call interface.

DEFEAT can be used at two basic user levels. At the first level, the user only implements an EA and tests this EA using one of the test problems contained in the DEFEAT benchmark functions repository through function calls to the system. At the second level, the user can also implement new test problems as well as an EA and test this EA on the new problem using the DEFEAT environment control mechanism to change the environment.

The basic components of DEFEAT are given in Fig. 1 and can be listed as follows:

- a tool to change the environment according to a predefined scenario
- a problem specific component in which the environment is defined through objectives, constraints and problem specific data (may also be implemented by user)
- a scenario creator to generate different change scenarios

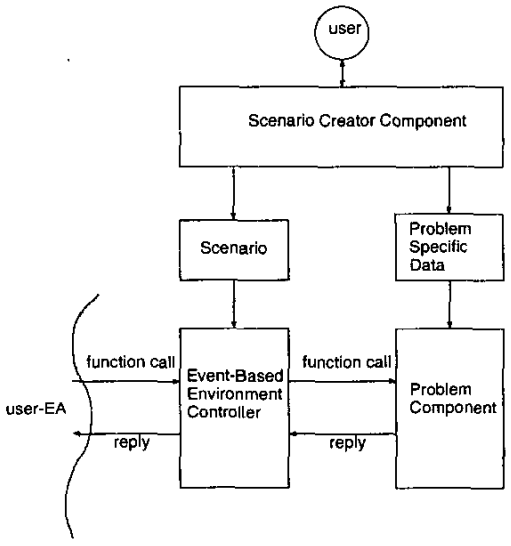

Fig. 1. Basic components of DEFEAT

[1]The source codes written in C are available for download as a zipped file from http://www.cs.itu.edu.tr/~etaner/DEFEAT.zip

The main motivation behind this level of abstraction is to provide a test framework as flexible as possible. Through the separation of the environment controller component and the problem specific component, it is possible to use the system with different problems. In the following subsections, the different components will be explained in detail, focusing on how they implement the desirable features of a test problem generator.

A. Main Components of DEFEAT

The interaction between DEFEAT's components, detailed in the following sections, can be seen in Fig. 2.

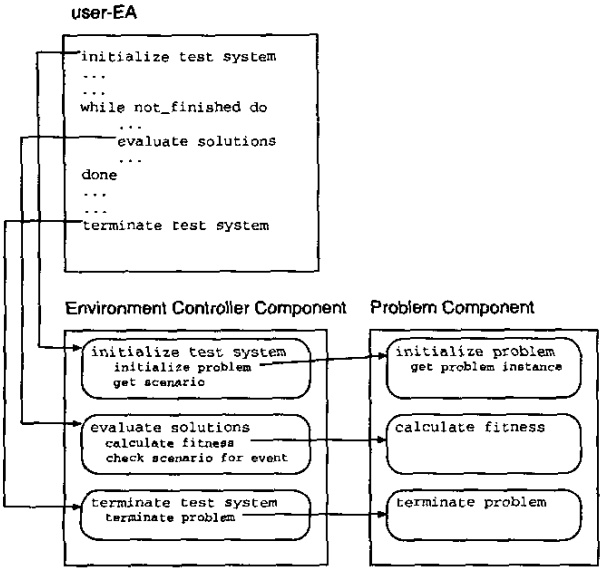

Fig. 2. Outline of the system

1) The Scenario Creator Component: The environment controller component works as an event based system, so the change instances are defined as events in a scenario. The scenario creator component is used to generate these scenarios. A typical scenario consists of the following information:

- base time unit for the framework clock (i.e. number of fitness evaluations or generations)
- total number of events
- a list of event times and the corresponding optimal solutions (if known) for these instances

It is possible to create scenarios defining only the event times and not providing any actual optimal solutions. This feature allows the extension of a test problem to a real world situation where the optimal solutions may not be known in advance.

The scenario is created by the scenario creator component through interactions with the user. Based on the user supplied preferences, the following types of change can be created either manually by the user or automatically by the scenario creator (assuming the optimum values for each change instance will also be included in the scenario):

- changes with low, moderate and/or high severity
- solutions oscillating between two values with fixed time intervals
- solutions oscillating between n predefined values with fixed time intervals
- periodic changes (constant time intervals) and random new solutions
- random time intervals and random new solutions
  - events occurring based on a uniformly distributed random time interval
  - events occurring according to a Poisson process, i.e. with exponentially distributed interarrival times between events

If no optimal solutions will be included in the scenario, the scenario creator only creates event times based on user preferences. The generated scenario is used by the environment controller component.

2) The Environment Controller Component: An event-based approach is used in this component where each discrete change instance is treated as an event. Events occur following a predefined scenario providing the times of scheduled change instances. The scenario is created using a scenario creator tool providing the required flexibility to implement many different types of change as explained in the previous subsection. A framework clock keeps track of the time units that have passed in the EA so far. Changes can be implemented based on the number of fitness evaluations or the number of generations by setting the base time unit for the clock in the event creation system accordingly. The environment controller component currently provides three services for use in the EA:

- initialize the test system: The whole test system is initialized at this call. The current scenario is read in from a file and the required event-based system setup is made. At the end of this call, the user-EA receives the necessary information to set the chromosome length, the allele value range and the allele data type (i.e. integer or real)
- evaluate the solution: The user-EA sends an individual to the environment controller component which in turn passes it on to the problem specific component. The fitness calculated by the problem specific component is returned to the user-EA via the environment controller component. At each evaluation, the environment controller checks the scenario to see if there is a scheduled event for that time unit. If an event should occur, i.e. it is time for an environment change, the environment controller component makes the necessary changes in conjunction with the problem specific component to emulate the new environment. The user-EA is notified as to whether a change occurred or not and is given the information regarding the current chromosome length and the allele range in case they are affected by the change. The environment controller notifies the user-EA of a change in order to be compatible with many different types of EA implementations since some of them may need to detect the change or be notified of the change.
- terminate the test system

3) The Problem Implementation Component: This component is where the actual problem specific functions are implemented. The information needed to set the chromosome length and allele range in the user-EA are acquired from a problem definition resource by this component. The fitness evaluation method is also defined here.

If the optimal solutions are not given in the scenario file, the problem implementation component has to keep track of the state of the system and incorporate the necessary mechanisms and settings to change the environment when instructed to do so by the environment controller component.

Depending on the level of the user, either pre-implemented problem definitions can be used or new problems can be easily defined. The only restriction in defining a new problem is to conform to the function call interface of the test framework. This component communicates only with the environment controller component through the following calls:

- initialize the problem: When the environment controller component wishes to initialize the problem, the problem specific data are input to the problem specific component and the environment controller component is notified of the current settings for the problem instance which determine the actual chromosome length, allele value range and allele representation type for the user-EA.
- calculate fitness: When the environment controller wishes to have an individual evaluated, it sends the individual and, if available for the problem, the actual optimal solution for the current environment to the problem component which in turn evaluates the individual and returns its fitness.
- terminate the problem

4) The User-EA: The user-EA is not a part of the framework. It is where the actual evolutionary algorithm is implemented. The user-EA interacts with the environment controller component using the services defined in section IV-A.2. Due to the abstraction provided by the service call interface between the user-EA and the environment controller, the user can experiment with different types of EA implementations. The user-EA needs to learn the gene representation type which shows the encoding for the genes as real or integer, the chromosome length and the allele range from the environment controller component. The outline of an example user-EA implementation that uses the service calls is given in Fig. 3. In this example outline, it is assumed that on detection of a change, the user-EA makes the necessary adjustments (e.g. increases mutation rate or randomly initializes some individuals, etc).

B. The Implemented Problem Instances

For the current release, there are four problems included in the repository within the framework. These problems are the dynamic 0/1 knapsack problem, two versions of a simplified dynamic load balancing problem (with and without optimum solutions in the scenarios) and the moving peaks benchmark problem [3].

```
initialize test system;
initialize EA;
while not_finished do
    evaluation of individuals;
    if environment_changed
        handle change;
    parent selection;
    recombination;
    mutation;
    survivor selection;
done;
terminate test system;
```

Fig. 3. Outline of an example user-EA

1) The Dynamic 0/1 Knapsack Problem: The 0/1 knapsack problem is an NP-complete problem which is defined mathematically as finding

$$max \sum_{i=1}^{n} v_i x_i \tag{1}$$

subject to the weight constraint

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

where x_i are variables that can be set to either 0 or 1 and n is the problem size. The weight constraint W can be enforced by way of a penalty factor on the fitness value. If a solution is overweight then it is penalized by a factor proportional to the amount the actual weight is more than the maximum allowed weight. In the dynamic 0/1 knapsack problem, the weight constraint W is changed. In the example 0/1 dynamic knapsack problem included in the framework (as previously defined in [16] by one of the authors), the fitness of an individual is calculated using a penalty approach.[2] If a solution satisfies the weight constraint, no penalty is applied and the fitness is calculated as given in Equation 3.

$$\sum_{i=0}^{n} v_i x_i \tag{3}$$

If the solution is overweight, the penalized fitness of the individual representing the overweight solution is calculated using Equation 4. This equation ensures that overweight individuals are severely penalized and have fitness values between 0 and 1.

$$fitness = \frac{(\sum_{i=0}^{n} v_i x_i) * (1 - \frac{|(\sum_{i=0}^{n} w_i x_i) - W|}{\sum_{i=0}^{n} w_i})}{\sum_{i=0}^{n} v_i} \tag{4}$$

[2]For best performance in solving knapsack problems, other approaches for handling infeasible individuals can be used. The penalty function provided here is given only as an example.

2) The Simplified Dynamic Load Balancing Problem: A simplified model of a dynamic load balancing of jobs on processing units (PU) is implemented (as previously defined in [14] and in [15] by one of the authors). The objective is to minimize the total load imbalance throughout the system. For simplification, the following assumptions are made about the system without loss of generality:

- All PUs in the system are equipped with the same type of resources with different capacities.
- All jobs may be migrated.
- There are no constraints. The total load on a PU may exceed its capacity.
- At the beginning of job execution, the average resource (CPU, I/O, Memory) requirements per unit time for each job are determined randomly. It is assumed that actual resource requirements do not deviate too much from the average values. The load value assigned to the job is a function of average requirements per unit time for all types of resources.
- The migration costs are ignored.

There are six possible types of change:

- arrival of a new job
- completion of a job
- change in the load of a currently existing job
- addition of a new PU
- removal of a PU
- a change in the capacity of a currently existing PU

The fitness of an individual shows how balanced the load distribution represented by the phenotype of that individual is. For each PU, the amount of load for that PU under ideal conditions, which will be called *ideal load* (IL), is calculated using the current total load in the system and the capacities of each PU as given in Equation 5.

$$IL_i = \frac{Total\,System\,Load}{Total\,System\,Capacity} * Capacity_i \qquad (5)$$

The *load imbalance* (LI) of a PU is the absolute value of the difference between the *actual load* (AL) of a PU and its *ideal load* (IL). To normalize the total imbalance value, it is divided by the current total load in the system. The normalized load imbalance is given in Equation 6.

$$LI_N = \frac{\Sigma_i |AL_i - IL_i|}{Total\,System\,Load} \quad i = 0, 1, ..., NoOfPUs \quad (6)$$

The fitness value f for an individual is calculated as in Equation 7:

$$f = \frac{1}{1 + LI_N} \qquad (7)$$

3) The Moving Peaks Benchmark Problem: To demonstrate the flexibility of the proposed framework, the moving peaks benchmark function developed by Branke [3] is also included. The moving peaks benchmark provides the EA with function calls to change the environment.[3] However, this approach requires that the EA explicitly initiates a change in the environment. In DEFEAT, a wrapper for the moving peaks benchmark is implemented and through this interface, the environment controller component interacts with the benchmark, implementing previously defined scenarios. So for this problem, the wrapper and the moving peaks benchmark together form the problem implementation component of the test framework. The source code and a detailed explanation of the moving peaks benchmark can be found at [18].

V. Sample Use of DEFEAT

Two different modes of use for DEFEAT are given below. Two versions of the simplified dynamic load balancing problem is implemented for the demonstrations. Firstly, a more controlled but restricted implementation will be explained where the optimum is known for each change instance. Secondly, using DEFEAT with still a simplified but a more realistic dynamic load balancing problem where the optimum is not known at each change instance will be explored.

For both examples it will be assumed that there are initially 10 jobs and 4 PUs in the system. An example initial resource definition file to be used by the problem component is given in Fig. 4.

```
jobs 10
PUs 4

# info on initial job resource requirements
info jobs
783
2110
173
3279
218
1205
413
1402
1660
89

# info on initial PU capacities
info PUs
1263
896
2300
3050
```

Fig. 4. Example resource definition file for the dynamic load balancing problems

A. Usage Example 1

In the first example, the problem instance where the optimum at each change is known will be explored. Through the scenario creator, the user supplies the necessary input to generate an environment change scenario containing the change times and optimum solutions at each instance.

[3]The moving peaks benchmark provides other function calls for performance measuring. Since this feature is left to future releases of DEFEAT, these extra calls provided in the moving peaks benchmark are ignored in the current implementation.

Assume that an example log file generated by the scenario creator is as given in Fig. 5. It can be seen in the log file that the base time unit for the changes is generations and interarrival times of the change events are exponentially distributed with a mean of 250 generations. The probabilities for the occurrence of each change severity type is defined and the scenario creator automatically creates the events as given at the end of the log file. The initial optimum solution is determined randomly. In the solution, each location on the string shows the PU the corresponding job is assigned to for achieving the optimum allocation. The new solutions in the next change instances are determined by changing the previous solution at a number of locations (determined randomly) as defined by the severity of the change. This method of changing the optimum solution corresponds to changes in the resource requirements of jobs, or changes in the capacities of PUs. Consequently, no new/finished job and no added/removed PU events are possible in this version of the problem. As a result of this simplification, on the user-EA side, the chromosome length and the allele value range remains constant at all times.

```
generate solutions = YES
input solution string length = 10
solution locus cardinality = 4
LOW SEVERITY= [0.0, 0.25)
MEDIUM SEVERITY= [0.25, 0.75)
HIGH SEVERITY= [0.75, 1.0]
total no. of events = 11
base time unit = GENERATIONS
event generation method = AUTOMATIC
interval_type = RANDOM
distribution = EXPONENTIAL (mu=250)
change dynamics = RANDOM
 LOW SEVERITY change probability = 0.333
 MEDIUM SEVERITY change probability = 0.333
 HIGH SEVERITY change probability = 0.334
events:
0       -       0,2,0,1,3,2,1,0,0,3
286     LOW     0,2,1,1,3,2,1,0,0,3
698     HIGH    3,2,3,2,0,0,3,1,0,1
755     LOW     3,2,3,1,0,0,3,1,2,1
951     LOW     2,2,3,1,0,0,3,1,2,1
1135    HIGH    3,0,0,0,3,0,1,2,1,3
1248    MEDIUM  1,2,0,0,2,3,1,0,1,3
1299    HIGH    2,1,2,0,2,1,3,3,3,0
1634    LOW     2,1,2,0,0,1,3,1,3,0
1700    MEDIUM  2,3,1,2,0,3,3,1,0,1
1850    HIGH    0,2,3,0,1,2,2,3,2,1
```

Fig. 5. Example log from scenario creator component (including solutions)

The actual scenario file used by the environment controller is as given in Fig. 6. This scenario file is generated by the scenario creator component using the settings shown in the log file and it only contains the event times and optimum values for each change instance.

On the user-EA side, as a result of initializing the test system,

- the chromosome length is set to 10 as determined by the number of jobs,
- allele value type is set to be integers,
- the allele value range is set to be between 0 and 3 as determined by the number of PUs.

For each individual evaluation request the user-EA makes, the environment controller checks the scenario to see if it is time

```
#EXPO(250), RND(equal prob)
#no of events
11
#base unit
Generations
#event info
#time   solution
0       0,2,0,1,3,2,1,0,0,3
286     0,2,1,1,3,2,1,0,0,3
698     3,2,3,2,0,0,3,1,0,1
755     3,2,3,1,0,0,3,1,2,1
951     2,2,3,1,0,0,3,1,2,1
1135    3,0,0,0,3,0,1,2,1,3
1248    1,2,0,0,2,3,1,0,1,3
1299    2,1,2,0,2,1,3,3,3,0
1634    2,1,2,0,0,1,3,1,3,0
1700    2,3,1,2,0,3,3,1,0,1
1850    0,2,3,0,1,2,2,3,2,1
```

Fig. 6. Example scenario file (including solutions)

for a change. If it is not, it passes on the current optimum value and the individual to be evaluated to the problem component. In the problem component, the ideal loads for the PUs are calculated by using the optimum solution to find the ideal distribution of the jobs. The actual initial capacities of the PUs are not used in this version of the problem. It is assumed that the distribution supplied as the optimum solution by the environment controller component is the ideal distribution which is based on the current capacities of the PUs in the system. The fitness of an individual is calculated based on this ideal load value as given in Equation 7. The fitness value and the values corresponding to the chromosome length and the allele range in the user-EA are returned to the environment controller component.

The environment controller sends the fitness evaluation result and the new chromosome length and allele value range to the user-EA as well as a notification whether a change occurred or not. In this version of the problem, since no changes affect the chromosome length and the allele value range, the user-EA can ignore these notifications.

At the end of the user-EA run, the test system and consequently the problem instance are terminated. For this release of DEFEAT, performance assessment and statistical calculations are left to the user-EA.

B. Usage Example 2

In this second example, the problem instance where the optimum at each change is not known will be explored. In this version, all six types of change (new job, finished job, changed job resource requirements, added PU, removed PU, changed PU capacities) can be implemented. The problem component needs to keep track of the current environment, i.e. the number of jobs and PUs currently existing in the system and their corresponding resource requirements and capacities respectively. Through the scenario creator, the user supplies the necessary input to generate an environment change scenario composed only of change event times.

Assume that an example log file generated by the scenario creator is as given in Fig. 7. It can be seen in the log file

that the base time unit for the changes is again generations and interarrival times of the change events are exponentially distributed with a mean of 250 generations. However there is no information regarding the optimum solution at each change interval.

```
generate solutions = NO
input solution string length = 10
solution locus cardinality = 4
total no. of events = 7
base time unit = GENERATIONS
event generation method = AUTOMATIC
interval_type = RANDOM
distribution = EXPONENTIAL (mu=250)
events:
0
286
755
951
1248
1299
1700
```

Fig. 7. Example log from scenario creator component (without solutions)

The actual scenario file used by the environment controller is given in Fig. 8. This scenario file is generated by the scenario creator component using the settings shown in the log file and it only contains the event times.

```
#EXPO(250)
#no of events
7
#base unit
Generations
#event info
#time
0
286
755
951
1248
1299
1700
```

Fig. 8. Example scenario file (without solutions)

Since the resource definitions are the same as in the previous example, on the user-EA side, the same initial parameter settings for chromosome length, allele value data type and allele value range take place as a result of the initialize test system call.

For each individual evaluation request the user-EA makes, the environment controller again checks the scenario to see if it is time for a change. Then it passes on the individual to be evaluated and a flag to indicate whether a change occurred or not to the problem component. If it is not time for a change, as indicated by the value of the flag, in the problem component, the ideal loads for the PUs are calculated by using the current total load in the system and the capacities of all the PUs as given in Equation 5. Fitness of an individual is calculated based on this ideal load value and returned to the environment controller component. If it is time for a change, the problem component initiates a change and makes the appropriate adjustments (e.g. total load in system, number of jobs and PUs, etc). The different possible types of change and their corresponding occurrence probabilities are all defined and implemented in the problem component. The problem component reports the current number of jobs and PUs to the environment controller.

The environment controller sends this information (the chromosome length and the allele value range on the user-EA side) to the user-EA along with the fitness of the individual and a notification that a change has occurred. In this version of the problem, some types of change (new/finished job and added/removed PU events) affect the chromosome length and the allele value range, so the user-EA needs to check these notifications to make the necessary adjustments if needed.

Even though the environment controller sends the new chromosome length and the new allele value range to the user-EA, it does not include any information such as which job is finished or which PU is removed. To overcome this limitation, some changes need to be made to the function call interfaces both between the user-EA and the environment controller and also between the environment controller and the problem component. However, this would cause the framework to be more problem specific and will compromise its flexibility.

As in the previous version of the problem, at the end of the user-EA run, the test system and consequently, the problem instance are terminated.

VI. DEFEAT as a Dynamic Environment Test Framework for EAs

Through the implemented components, DEFEAT satisfies the desired features of a good test problem generator as can be seen below:

- Through the scenario creator component, different types of controlled change instances, having the desired different change severities and desired types of change time intervals, can be implemented.
- Through the problem implementation component, different problems can be defined. The actual definition of the problem determines how extensions to real-world problems can be made.
- Since the test framework is used through basically three function calls from within a user-EA, it is very easy to use.
- The scenario creator program, especially when used in the automatic mode, acquires the necessary settings from the user and creates scenarios accordingly, requiring no other implementation changes. The problem implementation component needs to be rewritten to implement different problems, however it is quite easy to integrate the new problem into the framework due to the well defined interface between the environment controller and problem implementation components.
- The whole test environment source code is currently developed using ANSI C so that it can be built and used on a variety of platforms.

VII. Conclusions and Future Work

In this paper, an event-based test framework for EAs in dynamic environments (DEFEAT) is presented. DEFEAT is made up of components that separate the user-EA from the environment change control mechanism and the problem implementation, making it easy to work with various problems and problem instances. It also includes a repository of implemented benchmark functions. The event-based approach and helper tools such as the scenario creator make it a suitable test framework that meets the requirements expected from such test environments. Through these features it can form a basis for making healthy comparisons between various EA approaches and implementations on dynamic environments.

DEFEAT is still at the very early stages of its development. There are some features to be included in future releases. Currently, performance measures and statistical calculations are left to the user-EA. However, these will be incorporated into the main environment controller component for standardization in performance comparisons. More benchmark problems (as needed by the users of DEFEAT) will be implemented and included. Random events currently occur either according to the uniform distribution or the Poisson process. However, other distributions will be implemented as needed. The first release of DEFEAT is implemented in C for a more widespread usage among the EA community, however the authors are considering to continue with further implementations using an object-oriented language to provide a cleaner interface and more flexibility for further improvements.

References

[1] Baeck T., "On the Behavior of Evolutionary Algorithms in Dynamic Environments", IEEE International Conference on Evolutionary Computation, pp. 446-451, IEEE, 1998.

[2] Branke J., "Evolutionary Algorithms for Dynamic Optimization Problems - A Survey", Technical Report 387, Institute AIFB, University of Karlsruhe, 1999.

[3] Branke J., "Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems", Proceedings of Congress on Evolutionary Computation CEC-99, pp. 1875-1882, IEEE, 1999.

[4] Branke J., Kaussler T., Schmidt C., Schmeck H., "A Multi-Population Approach to Dynamic Optimization Problems", Adaptive Computing in Design and Manufacturing 2000, Springer, 2000.

[5] Branke J., "Evolutionary Optimization in Dynamic Environments", Kluwer Academic, 2002.

[6] Gaspar A., Collard P., "Time Dependent Optimization with a Folding Genetic algorithm:, International Conference on Tools for Artificial Intelligence, pp. 207-214, IEEE Computer Society Press, 1997.

[7] Lewis J., Hart E., Graeme R., "A Comparison of Dominance Mechanisms and Simple Mutation on Non-Stationary Problems", Proceedings of Parallel Problem Solving from Nature, vol. 1498 of LNCS Springer Verlag, 1998.

[8] Morrison R. W., De Jong K. A., "A Test Problem Generator for Non-Stationary Environments", Proceedings of Congress on Evolutionary Computation CEC-99, pp. 2047-2053, IEEE, 1999.

[9] Morrison R. W., De Jong K. A., "Triggered Hypermutation Revisited", Proceedings of the 2000 Congress on Evolutionary Computation CEC-00, pp. 1025-1032, IEEE, 2000.

[10] Ng K.P., Wong K.C., "A New Diploid Scheme and Dominance Change Mechanism for Non-Stationary Function Optimization", Proceedings of the Sixth International Conference on Genetic Algorithms, pp. 159-166, Morgan-Kaufmann, 1995.

[11] Ryan C., "Diploidy without Dominance", 3rd Nordic Workshop on Genetic Algorithms, pp. 63-70, 1997.

[12] Smith R. E., "Diploid Genetic Algorithms for Search in Time Varying Environments", Annual Southeast Regional Conference of the ACM, pp. 175-179, 1987.

[13] Smith R. E., Smith J. E., "New Methods for Tunable, Random Landscapes", Proceedings of Foundations of Genetic Algorithms 6, pp. 47-68, Morgan-Kaufmann, 2001.

[14] Uyar A. S., Harmanci A. E., "Application of an Improved Diploid Genetic Algorithm for Optimizing Performance Through Dynamic Load Balancing", Advances in Simulation, Systems Theory and Systems Engineering, Electrical and Computer Engineering Series, pp. 423-428, WSEAS Press, 2002.

[15] Uyar A. S., Harmanci A. E., "An Adaptive Domination Map Approach for Multi-Allelic Diploid Genetic Algorithms", in GECCO 2003: Genetic and Evolutionary Computation Conference Late Breaking Papers, pp. 291-298, 2003.

[16] Uyar A. S., Harmanci A. E., "Comparison of Domination Approaches for Diploid Binary Genetic Algorithms", Applications and Science in Soft Computing, Advances in Soft Computing Series, pp. 75-80, Springer, 2004.

[17] *http://www.aifb.uni-karlsruhe.de/~jbr/GECCO2003/*, "Evolutionary Algorithms for Dynamic Optimization Problems Workshop" part of Genetic and Evolutionary Computation Conference (GECCO-2003), Chicago - USA, July 12, 2003.

[18] *http://www.aifb.uni-karlsruhe.de/~jbr/movpeaks/*, The Moving Peaks Benchmark