On the modelling of evolutionary algorithms

P. A. N. Bosman, D. Thierens

UU-CS-1999-11

On The Modelling Of Evolutionary Algorithms

Peter A.N. Bosman
peterb@cs.uu.nl

Dirk Thierens
Dirk.Thierens@cs.uu.nl

Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

Creating general systems and development environments for evolutionary algorithms (EAs) brings many advantages in both development and usage. A few of them have been built with different views on issues that are at the heart. In this paper we show that some of these issues are both common and nontrivial. With at the background a new system called *EA Visualizer*, we provide general solutions to those issues. Furthermore, we discuss the requirements and provide good insights in the modelling of EAs. We show how to overcome hard problems so as to achieve a high level of useability and flexability.

1 Introduction

In both implementations as well as theories, a more explicit and mechanical setting has been rising of late in Evolutionary Computation. Some repositories and libraries have been created [2, 4, 5, 6, 7, 8], providing frameworks in which EAs can be developed with both more ease and uniformity. In research lessons are learned from prior experiments and enhancements are made to design EAs that are competent.

As proof is hard to come by when working with EAs, visualizing results is essential to being confident about theories. When the evolutionary process is made interactive and the viewing continuous, the best of control is achieved over the execution and theories can be stressed very well. Furthermore, a general framework for developing EAs results in both ease and uniformity. Thus, a very useful tool is a system in which

1. information resulting from an EA, both during its run and at its termination can be visualized continuously as well as interactively and
2. development of new algorithms is done by coding new instances of components that are placed in a general framework.

The latter argument keeps us from having to write code for the entire EA. It allows us to merely define instances of parts from an at the outset defined algorithm. Alongside this we define views that visualize the required information. The instances can then be put together to create an actual EA and the views can be placed for visualization. This way we can both visualize theories in a powerful way and disregard a large amount of overhead in implementing a new EA.

Issues such as these have led to the development of a JAVA program named *EA Visualizer*. This paper contains information on the requirements and shows what problems are likely to be encountered in the creation of any such system. These problems are not uncommon or trivial. Using the design of the *EA Visualizer*, we also present possible solutions, moving beyond methods and approaches provided by other studies [3, 5].

The remainder of this paper is organized as follows. In section 2 we describe the requirements for creating such a system. Section 3 presents our approach to the modelling of EAs. In section 4 we then describe the problems that one encounters and present solutions as used in the *EA Visualizer*. Section 5 briefly introduces this new system, after which in section 6 we propose topics of further possible research. Finally, in section 7 we present our conclusions.

2 Requirements analysis

Creating large software systems is always done in a number of steps. Here we restrict ourselves to the requirements analysis. We discuss the three most important aspects, but also briefly point out other topics of importance in the creation of a general EA system.

2.1 Modelling EAs

First of all, we must question ourselves what EAs are. One approach, as presented by Leonhardi et al. [5], is to go from optimization and leave details on EAs unspecified. We then move to design a general system in which the algorithms are almost unrestricted. For instance we would then only have representations and operators. The operators are not specific such as a *Recombinator* or *Mutator*, but of a general type (eg. operator or not). An algorithm would then be defined as a sequence of operators to apply to a population or its members. In such a case, we move away from the modelling of EAs since the EAs are still to be programmed without a general model. Our approach is to look closer at evolutionary computation and take out the most general aspects. In this way we allow to define EAs through components that are common. We thereby put the focus on EAs instead of optimization, clarifying what EAs are in the system and how they can be composed.

A modular decomposition of EAs results in a description that consists of components. The system uses these components and passes the required data between them. By doing this, a high level of expandability is created as each component can have a multiple of instances. The user never has to redefine the entire process, but only to implement new instances such as an actual mutation operator. Note that this is done independent of the other components. For usage, the desired instances are selected to be part of the EA without any changes to the structure of the system.

A drawback is that by specifying any general framework other than that of total freedom, the EAs that can be created are of a certain form. Thus we must ensure that the expressional power of the modular framework is great, which we elaborate upon in section 3.

2.2 Visualizing information

In order to establish a flexible and expandable visualization part as well, this part should be separated from the EAs as much as possible. This perspective leads to a Model–View–Controller (MVC) type of approach in which the model (the EA) is separated from the view (the visualization part) and the system itself is controlled by the controller (the main system). For expandability, we require a modular structure based upon separate views for the visualization part.

Establishing *continuous* visualizations means that the system will have to transfer information from the model to the controller after each generational step. Every generation, all views should be updated by the system with the new data. It is then left to the views to decide if they need to display new information.

The level of *interactivity* with the system poses a problem. Allowing a direct influence on the EA through the views makes the MVC separation harder to establish and presents the user with more tricky system details. The viewing part should however receive all information from the model as we cannot know in advance what a view is going to visualize. This information includes the instances for the components in the EA. This allows to interactively alter EA parts, as long as the views can receive input. By doing so, the possible level of interaction extends from altering view characteristics to having an influence on the current EA without seriously harming the MVC property.

2.3 Running EAs

Because of their random aspect, EAs belong to the class of probabilistic algorithms. We can never rely on the performance of these algorithms in one single run. Experiments with EAs should therefore always be performed a multiple of times and the results combined.

Not only are continuous and interactive visualizations a great tool to learn about and research EAs, they are also a way to quickly satisfy intuitions. Using the single run version also prevents wasting computing time over a multiple of runs on wrong settings. Still for a thorough investigation, the user will want to be able to perform a multiple of runs. Therefore, any general system for EAs is not complete without the capability to run an EA a multiple of times.

At the same time it is common practice to vary parameters or strategies. Thus the availability of multiple runs should be accompanied by the possibility to run algorithms with different settings. However, as we shall see in section 4, automating this is quite complex.

2.4 An overview of the requirements

So far we have analysed what is fundamentally required in order to build a general system for EAs. We now summarize the requirements, introducing other topics of importance as well.

1. The system should have a modular structure. The most flexible parts must be decomposed so that they are easily expanded. This means that
   (a) visualization is done by views that are implemented as separate classes just as
   (b) the instances for the components of the EA are implemented separately.

2. The views should be capable of processing user input at the least by using a mouse pointing device, making the visualization interactive.
3. The system should transfer everything that could be of interest from the model to the views. This allows the interaction to have a direct influence on the EA and allows views to be general or specific.
4. The EA should be described by a general framework that is a decomposition of the evolutionary process. It must hold a great amount of expressional power, but needs not to be defined for all specific cases, making it too complex.
5. The system should be capable to run EAs a single run as well as a multiple amount of runs. In each of the multiple runs, a multiple of combinations of parameters or strategies must be settable.
6. The system should be self–contained through the containment of an easy to use extensive help system. This help system must be expandable for every other part of the system that is expandable.
7. The implementation should be such that administration of all parts that can be expanded is done in a mechanical way so that it can be automated, leading to an editor version that allows for easy editing and expanding of the system.
8. The implementation should be done in a language that will allow the system to run on as many different platforms and operating systems as possible. This will allow for users working on different platforms to exchange parts of the system.

In most systems, the larger obstacle for quick usage is that coding is still required in order to get something working [4, 5, 6, 7, 8]. The use of GUIs as a system would resolve this problem, which was acknowledged for instance by L. Dekker [2, 3]. None of these systems however have a fully integrated use of such GUIs through an editor as well as the running system. Such integration greatly improves useability. Most systems have no general view part, which prohibits a quick gain in insights. Almost all systems are coded in only partially portable languages, which prohibits general use. Finally, there is to our knowledge no other system available with a general multiple runs facilitation.

3 Decomposition of EAs

We are to create a general framework in which we identify components that have a clear and distinct functionality. The decomposition we present here has been implemented in the *EA Visualizer* and has been found to be a powerful one. Specific details on the derivation can be found elsewhere [1]. Here we only present this decomposition to get a feel for what such a modelling looks like. Note that it is possible for a system with other goals to have a different decomposition.

Figure 1 shows an intuitive description of the generational step in EAs in our decomposition. Application of components such as *Mater*, *Recombinator* and *Mutator* are classical. The implementation of the selection mechanism is achieved by two selection phases and one replacement phase. We have a *Replacer* component that merges in some way the contents of the current population with a collection of offspring genomes. We also have a *Selector* component, which is used twice. It is used once at the beginning to select the parent genomes. It is then used again at the end to select the genomes that are to finally survive. This leads to only one new component, of which two possibly different instances are used.

When denoting the algorithm depicted in figure 1 by $evolve(P(t))$ where $P(t)$ is the population at generation t, the full meta algorithm according to the decomposition can be stated as in figure 2. Not all components can be derived immediately from the above. As the description so far only regards the dynamics, we might overlook important parts in the background such as the fitness function or the genotype. Identifying these components, the decomposition used in the *EA Visualizer* consists of the twelve components presented in figure 3.

The proposed decomposition is quite expressionally powerful. Elsewhere [1] it has been shown that not only does this decomposition allow for a modelling of the classic GA, it is also well suited for algorithms such as the mGA, the fmGA, the GEMGA and the LLGA as well as other evolutionary approaches such as MIMIC, evolution strategies and problem specific variants such as approaches for the TSP. Furthermore, commonly used add–ons for EAs can easily be incorporated such as elitism, crowding, preselection and sharing.

4 Common problematic issues

In this section we regard common problematic issues one is confronted with in design and implementation. Even though the issues are general to all systems, the solutions presented are exactly the ways in which they are implemented in the *EA Visualizer*.

4.1 Dependencies

The most common problem is the combinations of instances for the components. This problem arises be-

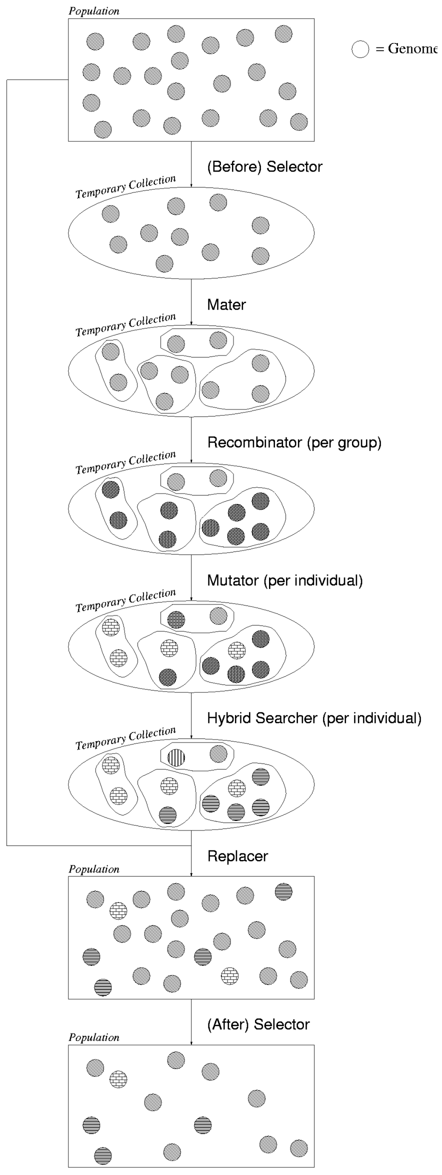


Figure 1: Fine grained decomposition of the generation step in EAs.

```
$t = 0$
$initialize(P(t))$
$evaluate(P(t))$
**while not** $terminate(P(t))$ **do**
    $P(t+1) = evolve(P(t))$
    $evaluate(P(t+1))$
    $t = t + 1$
**od**
```

Figure 2: Coarse grained decomposition of EAs, the full algorithm.

Genotype	Population	Fitness Function
Similarity	Selector	Mater
Recombinator	Mutator	Hybrid Searcher
Replacer	Terminator	PRNG

Figure 3: The twelve components in the decomposition used in the *EA Visualizer*.

cause some components use others. In most decompositions, this means that the operators and data structures pose such a problem. In our framework in section 3, there is for instance a problem between the *Recombinator* and the *Genotype*, because we can't use one–point crossover for binary strings on TSP tours. Neither can we use a binary mutator on an evolution strategies tuple. This requires us to restrict the construction of EAs to legal combinations of instances.

A solution to this problem is to identify some of the components as *dependency imposing*. Such a component is used by another component. What instances for a component are available thus depends on what kind of dependency components are installed. In our decomposition these components are *Genotype*, *Fitness Function*, *Population* and *Similarity*. Respecting this dependency imposing stucture, the user can be forced to select only legal combinations. For each instance information is stored regarding what instances for each dependency imposing component this instance is allowed to be selected with. Every time an instance is selected for one of the dependency imposing components, a filter can be called upon to filter the lists of available instances for all components.

4.2 Parameters

It is common practice to be able to vary for instance the population size and the string length for the genomes. For each instance we should therefore realize that it might have parameters for which values need to be entered. At this point we should separate the notion of *parameter* from *parameter value*. A *parameter* is the structure that defines what values can be set under what name. It can be compared to a *typed variable* in programming languages. The actual *value* for it is what is used at runtime. So for a *Selector* instance a parameter named *selection size* could be of type integer, whereas a value for it could be 100.

For each instance, the amount and type of parameters must be known. For a uniform system, we require a general parameter structure. The issue is to separate the instances from their parameters. As we shall see lateron, only through such loose coupling we are able to solve more advanced system flexability issues.

Our approach is to create *parameter components* that can be used by both the system and by the instances. For any instance, the parameters are specified as a list of parameter components. The system knows how to display them and once the values are entered, they can be shipped to the instance and there set.

4.3 Multiple runs

The system must be capable of running EAs a multiple of times. In our general setting however, facilitating this neatly is complex. At first glance, this might not seem to be the case because making a self coded GA run iterated is no more difficult than adding another loop to the code. This is however required for each type of test anew whereas a general model will prevent this. Furthermore, constructing a general approach helps in finding solutions to whatever loops required when coding them directly.

The simplest approach for the general case is to allow the user to create an amount of single run algorithms that are consequetively run a specified amount of times. This is far from desirable as it happens all too quickly that we wish to test all population sizes between 2 and 200 with a step of 4, which would require the definition of 49 EAs. Furthermore, we have then not even varied other settings as well.

A much better approach is to allow for the varying of settings. The parameter values must then be varyable so that the influence of a parameter can be inspected. In order to test different strategies, we must be able to test different instances for a single component. We must however not forget that we have *dependency imposing* components. This implies that we cannot test a multiple of instances for those components in a single specification. Otherwise we might get inconsistencies within the dependencies that are imposed. Also, we might want to enumerate sets of parameters or instances simultaneously (for instance amount of genomes to select and population size). Thus we need to be able to avoid crossproducts between settings.

To be able to install a multiple of instances for the dependency imposing components, we can allow for a multiple of multiple runs to be specified. Each of these multiple runs can be run a specified amount of times for each combination of parameter values and instances. In each such multiple run, the instances for the dependency imposing components are fixed, but their parameters are varyable. The instances for the other components can be selected with respect to the dependencies now imposed. For each instance, a multiple of parameter values can be entered. Because the parameter structure was isolated, a special facilitation might be implemented to this end. For one such multiple runs the links can be specified, indicating what sets of parameters or instances must be enumerated simultaneously. This is nontrivial, as not all combinations are allowed to be linked as can be seen in figure 5. The structure for instances and parameters are shown in figure 4. It is obvious that the complexity of multiple runs is far beyond that of single runs.

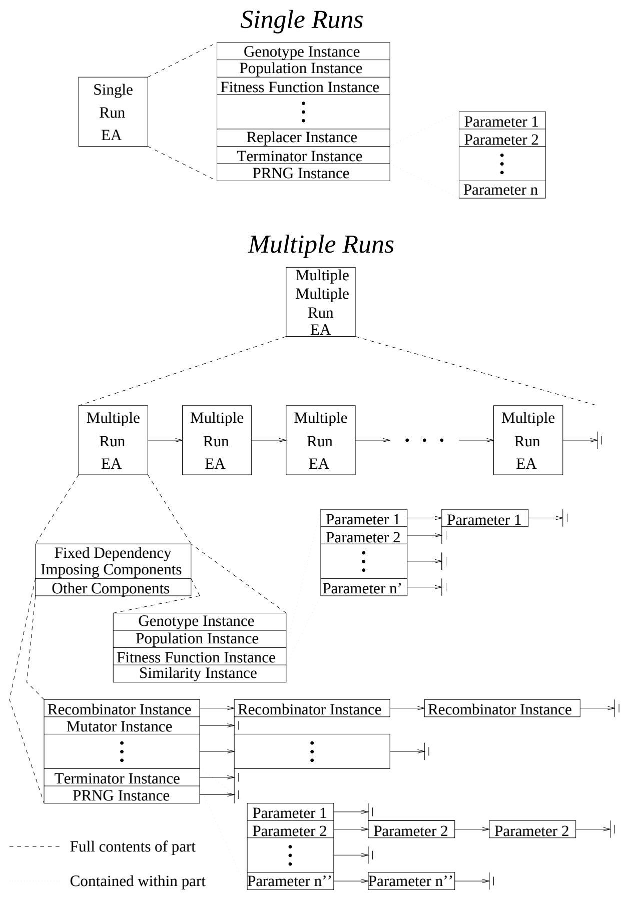


Figure 4: Structure of single and multiple runs with example instances and parameters.

At this point a special enumerator can be called upon to enumerate the settings in the proper manner. The system will create a new single run EA and run it for each set of settings. Creating the enumerator is a difficult task as we amongst other things have to respect the links that have been made. To this end, we propose a solution in which all that can be varied in a multiple run is appointed a counter. These counters are enumerated just as would be a row of bits, with the exception that every bit now has its own range of values. A specific setting for these counters then points to a setting for the parts in the EA.

Setting up the counters can be done in five phases. First, room is made to incorporate counters. For every component there should be a counter, as well as for each parameter for each instance of each component. For each set of linked items, we then appoint the same counter to each member. In a third pass, we pass out counters *only* for the sets of instances. This is required because we don't want to enumerate settings for component instance 2 if instance 1 is currently installed. Therefore we need to know what counters are associated with unlinked parameters only. The structure to be enumerated is shown in figure 5. The remaining counters are passed out in a fourth pass. In the final pass, the counter arrays are set up and the amount of combinations are counted. We have two types of counter arrays, namely a normal one and a *fast* one, where the *fast* counters are used for the parameters and their values. Note that for every time we increment the normal counters (which happens when we have run through the fast counters), we have to redetermine the fast counters and reset that array.

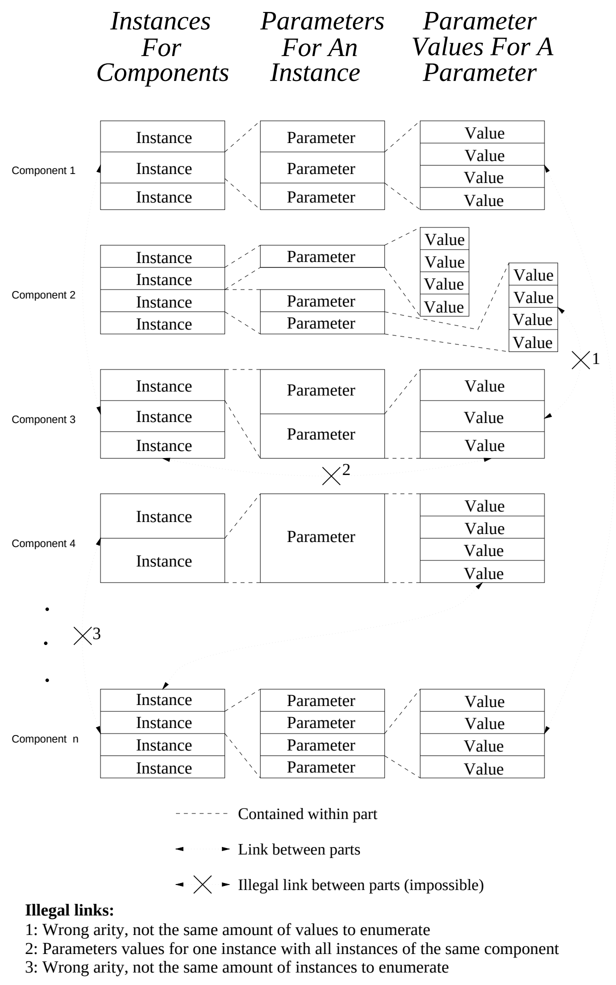
4.4 Visualizing information

Decoupling the visualization and the model allows us to create EAs without concerning ourselves with the gathering of data. This decoupling is established by shipping all information from the EA to a data processing part. We can now also disregard overhead within the EA for gathering statistical information for instance, which can now be done elsewhere. To this end we must send *enough* information, but this is addressed by creating a single package with all instances in the decomposition and their parameter values, the generation counter and other meta variables such as the chance at recombination and mutation. Also, all intermediate steps of the latest generation must be included (such as all genomes after recombination).

4.5 Useability

User interaction is in many systems text–based. The useability of the system can however greatly be improved by using *graphical user interfaces* (GUIs). It for instance allows us to present the user with lists of instances from which a selection has to be made. These lists can be kept consistent with the dependencies. This way EAs can be selected by simply "clicking them together", instead of having to write this out in a textfile by hand and to use a parser to report errors.

By allowing the user to select the EA in this way, we require to create instances at runtime. To keep the system expandable and automated, we require to have some tight structure in which this creation is done.



Instances For Components
Parameters For An Instance
Parameter Values For A Parameter
Component 1
Component 2
Component 3
Component 4
Component n
Instance
Parameter
Value
×1
×2
×3
Contained within part
Link between parts
Illegal link between parts (impossible)
Illegal links:
1: Wrong arity, not the same amount of values to enumerate
2: Parameters values for one instance with all instances of the same component
3: Wrong arity, not the same amount of instances to enumerate

Figure 5: The structures to be enumerated and an example of links between them.

This implies that we need some form of a *Creator Object* that is capable of creating instances. Once selections are made, the creator object knows what runtime objects to create for the selected items.

At this point, some parts come together as within such a creator class we can place the information on the parameters for the instances, as well as their dependencies. Thus, following this approach we can both offer the user a friendly way to use the system as well as enforce the solutions proposed in this paper.

4.6 Editing and expanding

Expandability is very important for the applicability of the system. In the neat most way, this requires to have a system editor. An undesirable approach would be to guide the user through the implementation parts and explain where to put what files and how to compile them within the system. Creating an editor requires software generation methods and strict protocols for automation. Based upon the neat approaches taken in this paper, creating such an editor becomes feasible.

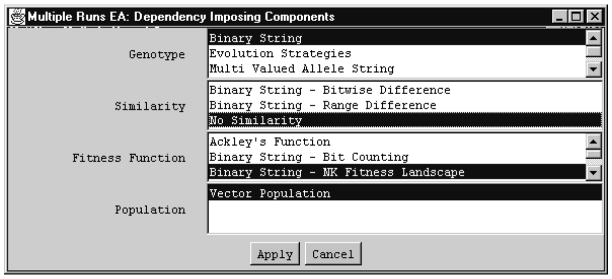
The system should allow the user to create new instances through coding them in some language. Also, the parameter structure and the view system need to be augmentable in this way. Furthermore for each new instance, the specification of parameters, dependencies and help pages must be facilitated. To make the expansion complete, the availability of new instances should be automated. We have already proposed *Creator Objects* in which such information is stored. Thus a system update consists of regenerating these objects so that they incorporate the new information.

5 An example: EA Visualizer

We now present the *EA Visualizer* [1], a system designed and implemented conform to the issues discussed in this paper. It can be tested online through an applet version or be downloaded at the *EA Visualizer* WWW site: http://www.cs.uu.nl/people/peterb/computer/ea/eavisualizer/EAVisualizer.html

We only give a feel for the *EA Visualizer* through a multiple runs example. The tests merely serve to be an example of how modelling aspects as discussed in this paper can come into being.

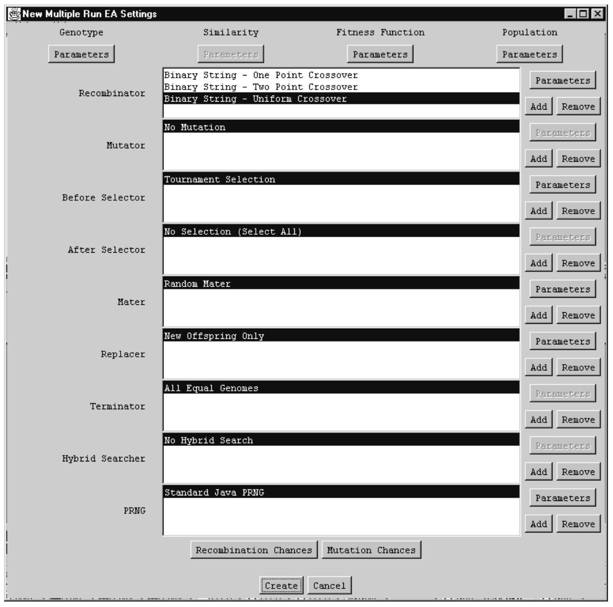
We set out to run a GA on a NK fitness landscape with three recombination strategies over increasing population sizes, combining results over thirty runs. We want to use tournament selection and use the offspring as the new population contents. One of the first steps in entering this in the *EA Visualizer*, is the selection of the instances for the dependency imposing components as can be seen in figure 6.



Genotype	Binary String; Evolution Strategies; Multi Valued Allele String
Similarity	Binary String - Bitwise Difference; Binary String - Range Difference; No Similarity
Fitness Function	Ackley's Function; Binary String - Bit Counting; Binary String - NK Fitness Landscape
Population	Vector Population

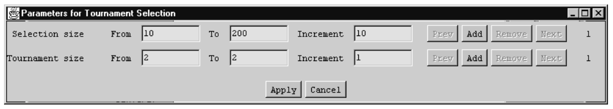
Figure 6: Selecting the instances for the dependency imposing components.

Next, instances can be added for the other components and their parameters can be set as depicted in figure 7. Figure 8 shows the settings for the tournament selec-



Recombinator	Binary String - One Point Crossover; Binary String - Two Point Crossover; Binary String - Uniform Crossover
Mutator	No Mutation
Before Selector	Tournament Selection
After Selector	No Selection (Select All)
Mater	Random Mater
Replacer	New Offspring Only
Terminator	All Equal Genomes
Hybrid Searcher	No Hybrid Search
PRNG	Standard Java PRNG

Figure 7: Entering the settings for the multiple runs EA.



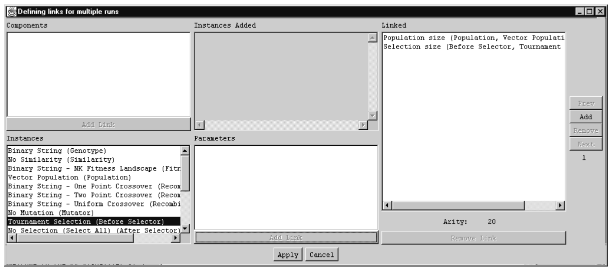
Selection size	From	10	To	200	Increment	10	1
Tournament size	From	2	To	2	Increment	1	1

Figure 8: Entering the settings for tournament selection.

tion operator. Just as for the population size, we enter to select amounts from 10 to 200 with steps of 10.

To let the offspring replace the contents of the population, we must select just as many genomes as the population has. Therefore, no crossproduct enumerations must be made between the population size values and the selection size values. This is entered by specifying the links as shown in figure 9. The lower left list contains all instances installed. The area on its right contains the parameters for each such instance once it is selected (and not linked yet). The linked items are shown on the right, which in this case are the population size and the selection size.

Having created the EA, we add some views for visualization. Here we only add some graphs with statistics. The running system is shown in figure 10. The upper two graphs show the fitness average and variance of the best fitness value after each of thirty runs. The lower two graphs show the average and variance values for the amount of fitness evaluation calls. In all graphs the solid, dotted and dashed lines represent one–point, two–point and uniform crossover respectively.



```
Linked:
Population size (Population, Vector Populati
Selection size (Before Selector, Tournament

Instances:
Binary String (Genotype)
No Similarity (Similarity)
Binary String - NK Fitness Landscape (Fitr
Vector Population (Population)
Binary String - One Point Crossover (Recom
Binary String - Two Point Crossover (Recom
Binary String - Uniform Crossover (Recombi
No Mutation (Mutator)
Tournament Selection (Before Selector)
No Selection (Select All) (After Selector)

Arity:    20
```

Figure 9: Linking parameters to avoid crossproduct enumerations.

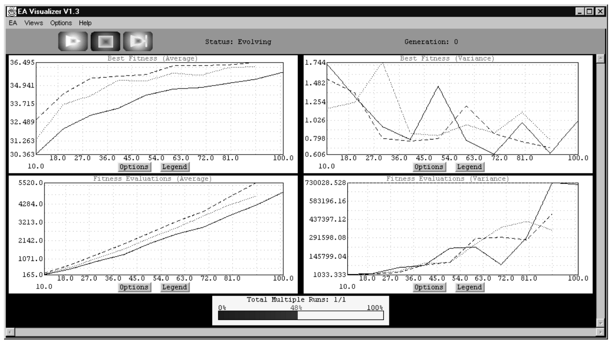


Figure 10: The *EA Visualizer* system while it is running the multiple runs.

6 Possible future extensions

Extensions can be made in many ways and the program is never finished. The system can for instance always be expanded to contain more instances for the components. We are however more interested in issues with respect to general modelling. There are some issues in that field that can still be visited.

The decomposition made does not explicitly allow for multiple strategies in multiple populations using migration schemes to exchange results. It is interesting to investigate how issues alter and what additional aspects might come into being when modelling this without forcing it into a more specialized decomposition.

It would be most interesting to look at parallelization of EAs. This would lead to additional modelling issues. It would provide a great advantage because more computing power could be applied. This issue deals with the modelling of EAs and less with the system itself. Aspects to the latter can thus remain unaltered as we have proposed a highly modular system.

7 Conclusions

Designing a general development environment for EAs brings many common and general problematic issues. We have presented solutions to the most important of these issues. Through our approaches it becomes feasible to create system aspects of great complexity. This in turn allows the user to work with EAs easier and in a more powerful and flexible way.

At the heart lies a good decomposition of EAs. For each new system this decomposition can be made according to the desired properties. We have presented a general framework in which most EAs can be modelled and in which it is convenient to think about them. Because of the modular approach we presented, all presented solutions can be used for common problems encountered when creating any of such useful tools.

References

[1] P.A.N. Bosman. A general framework and development environment for interactive visualizations of evolutionary algorithms in java and using it to investigate recent optimization algorithms that use a different approach to linkage learning. Utrecht University gradudation paper INF–SCR–98–15. http://www.cs.uu.nl/people/peterb/computer/papers.html, 1998

[2] L. Dekker. Q–game genetic algorithm development kit. http://www.cs.ucl.ac.uk/staff/L.Dekker/pubs/qgameman.ps, 1995

[3] L. Dekker and J. Kingdon. Development needs for diverse genetic algorithm design. In *Genetic Algorithms in Optimisation, Simulation and Modelling*, pages 9–26. IOS Press, 1994

[4] J.J. Grefenstette. A user's guide to genesis version 5.0. ftp://www.aic.nrl.navy.mil/pub/galist/src/genesis.tar.Z, 1990

[5] A. Leonhardi, W. Reissenberger, T. Schmelmer, K. Weicker, and N. Weicker. Development of problem–specific evolutionary algorithms. In A.E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving From Nature V*, pages 388–397. Springer–Verlag, 1998

[6] P.D. Surry and N.J. Radcliffe. Rpl2: A language and parallel framework for evolutionary computing. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Parallel Problem Solving From Nature III*, pages 628–637. Springer–Verlag, 1994

[7] H-M. Voigt, J. Born, and J. Treptow. The evolution machine manual 2.1. ftp://neisse.gfai.de/pub/software/em/em-man.ps.Z, 1991

[8] M. Wall. Galib: A c++ library of genetic algorithm components. http://lancet.mit.edu/ga/, 1998