2013 IEEE Congress on Evolutionary Computation
June 20-23, Cancún, México

A Setup Reduction Methodology from Lean Manufacturing for Development of Meta-heuristic Algorithms

Alejandro Teran-Somohano
Department of Industrial and Systems Engineering
Auburn University
Auburn, AL
ateran@auburn.edu

Alice E. Smith
Department of Industrial and Systems Engineering
Auburn University
Auburn, AL
smithae@auburn.edu

***Abstract*—We present an application of the Single Minute Exchange of Dies (SMED) methodology used in lean automotive manufacturing for the design and implementation of meta-heuristic algorithms. This methodology allows the design of algorithm implementations that are flexible and enables algorithm designers to modify the algorithm's configuration quickly and with a minimum amount of errors. This comes with little downside as measured by computational effort and engenders a wider variety of experimental scenarios to be tested.**

***Keywords*—SMED, Lean manufacturing, meta-heuristics, genetic algorithm**

I. INTRODUCTION

Meta-heuristic algorithms have become a widely used tool for solving difficult optimization problems. Meta-heuristics are distinguished from ordinary heuristics in that they are techniques that can be applied to a variety of problems. For this capability to be exploited successfully, however, it requires that an implementation of a meta-heuristic be configured for the specific problem it is trying to handle. So, for example, though a genetic algorithm can be used for solving different problems, it requires to be fine-tuned for each problem and perhaps even for problem instances. Furthermore, parameters might need to be adjusted if things such as the input data, its scales or its size change, even when dealing with exactly the same problem. That is, there is normally no such thing as a meta-heuristic implementation and configuration that can perform well across the board. Researchers, therefore, require extensive experimentation and exploration of different configurations in order to determine which are ``good" for the problem at hand. This takes up a lot of effort and work which could be reduced significantly with a good design of the meta-heuristic implementation. If the implementation is designed in such a way that allows for a quick change of configuration, the experimentation and exploration process can not only be performed more quickly and with fewer errors, it can even be automated.

The purpose of this work is to present a methodology that allows a developer to design and build a framework for the implementation of meta-heuristic algorithms that will significantly reduce the time and effort required to test different configurations, parameter values and even problems. It also intends to reduce the number of programming errors introduced by any changes done to the original source code. The methodology used in this work is based on the SMED (Single Minute Exchange of Dies) methodology from lean manufacturing that was created by Shigeo Shingo and used by Toyota in their car manufacturing plants.

Section 2 will briefly present and explain the SMED methodology, including any definitions and necessary terms. Section 3 will describe the problem we are attempting to address as well as develop the analogy between a manufacturing system and a computer program so that we can apply the SMED methodology. Section 4 explains how the steps of the SMED methodology were applied to a specific implementation of a genetic algorithm. Section 5 describes the experiments we set up to test our framework. Section 5.1 describes the metrics we used to measure the effectiveness of our proposed approach, section 5.2 presents the data that we used as part of our experimental phase, section 5.3 provides an overview of the specific experiments that we performed and section 5.4 describes in detail the results that we obtained. Finally, in section 6 we discuss our conclusions as well as future work that can be done based on the proposed framework.

II. BACKGROUND

Single Minute Exchange of Dies (SMED) is a methodology developed by Shigeo Shingo, a consultant working for the Toyota Manufacturing Company, which seeks to reduce the setup time of a machine to less than ten minutes [14]. This methodology is also known as “setup reduction” and both terms will be used interchangeably in this paper. It is considered one of the many lean tools that are used in the manufacturing industry to reduce waste and increase quality.

One of the main goals of the Toyota Production System (TPS) is flexibility [1]. SMED came into being within the automobile manufacturing industry, where large lots of parts were being produced repeatedly. SMED allowed Toyota to reduce its lot sizes, adding flexibility to its production system, which in turn reduced inventory and waste. In addition, this added flexibility in the system also enabled Toyota to produce highly customized cars and eventually led them to become the largest car manufacturer in the world [16].

SMED is performed in four sequential stages [13]:

978-1-4799-0454-9/13/$31.00 ©2013 IEEE

1. Stage 1: Determine the existing method

2. Stage 2: Separate the internal elements from the external elements

3. Stage 3: Shift the internal elements to external elements

4. Stage 4: Improve all elemental operations

Two definitions are of utmost importance in SMED: internal and external elements. Internal set-up elements are those operations that can be conducted only while a machine is stopped. External set-up elements are those that can be performed while the machine is still running [1]. By making all internal elements external, the amount of time needed for set-up is significantly reduced. Furthermore, the set-up process can be automated and, ideally, completely eliminated.

Fig. 1 presents a more detailed explanation of these stages.

Since Toyota's success in implementing the SMED methodology, its use expanded to other car manufacturers, the pharmaceutical industry [4], textile processing [11], ship building [14], electronics assembly [15] and it also has been proposed for many other sectors [10].

The idea of SMED seems to be applicable to any process that requires flexibility and personalization. One such process is the design and development of meta-heuristic algorithms. As was mentioned in the Introduction to this paper, meta-heuristics require a high degree of personalization in order to deploy their full potential for solving optimization problems. They also need to be flexible enough to address a variety of problems. Therefore, it seems that SMED ---or at least a variation of SMED--- can be useful for meta-heuristic algorithm design.

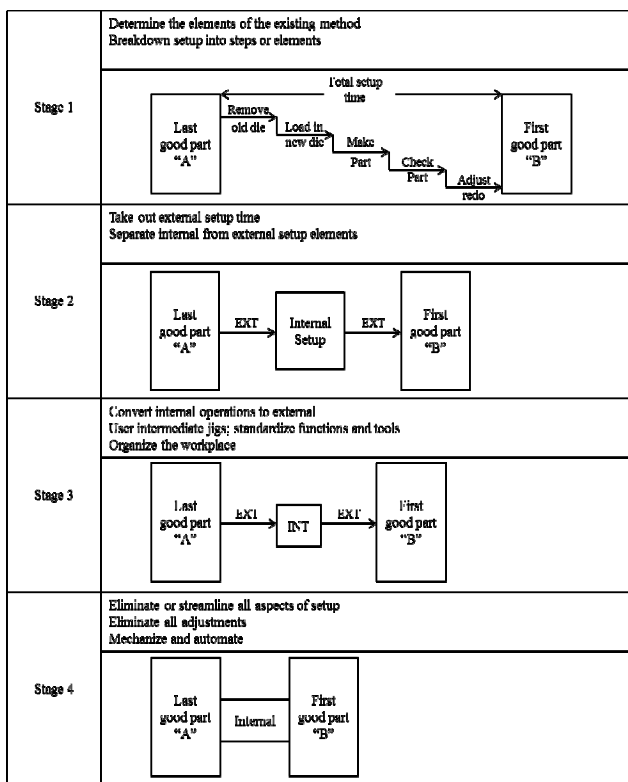

Fig. 1. Four stages of SMED [1]

The importance of design in the SMED methodology has been highlighted by [8] and [9]. Reference [5] lists a series of rules for designing new equipment that is specifically built to include quick set-up time capabilities. That is, there is a growing awareness that reducing set-up times is important and that it should be included as part of the design process. We propose to do this for meta-heuristic algorithm design.

Some of the ideas of SMED are already being applied in the software development field. Reference [6], for example, proposes implementing a lean manufacturing approach to information management systems. Several frameworks for designing optimization algorithms (including meta-heuristics) that seek to implement some of the ideas behind SMED are also available, such as Opt4j [7], ParadiEO [2] and EasyLocal++ [3]. However, their focus is often limited. For example, Opt4j is a framework for optimizing tasks where multiple heterogeneous aspects have to be optimized concurrently [7], ParadiEO is used for parallel and distributed optimization [2] and EasyLocal++ is limited to local search heuristics [3]. Reference [7] refers to some additional libraries and frameworks that are available but points out that they have the similar limitations to those that we have mentioned.

III. Problem Description

The problem we wish to address is that of excessive "set-up" time when configuring and experimenting with meta-heuristic algorithms. We intend to use the SMED methodology for solving this problem. The SMED approach will allow us to design meta-heuristic algorithm implementations that minimize (or potentially eliminate) these "set-up" times.

In order to do so, we need to come up with an analogy between manufacturing system design and software design, specifically, software that implements meta-heuristics. Once such an analogy has been established, we will be justified in using the SMED methodology to solve our problem.

In a lean manufacturing system, parts flow through a series of machines and workstations where they are processed and assembled into a final product. Ideally, lot sizes have been reduced to one. That is, each part can, potentially, be different from the one before it. In a way, each part has a ``unique'' configuration. In our system, a specific run of the algorithm is equivalent to a part. Therefore, the set-up time is the amount of time between one run and another. Each run can —and often will— make use of different parameter values and even different settings of the algorithm.

At each machine, a part is processed with fixtures and machining tools that correspond both to the specific part and the specific process it is going through. The use of SMED has enabled the fixtures and components of a single machine to be quickly switched to those that are needed to process the incoming part. We can think of the specific meta-heuristic we are using as the process through which a given part must go through. The program that executes the algorithm (the main function) is equivalent to a machine and its different modules or functions are the fixtures and machine tools.

There are two more terms that need to be included in our analogy. We define as *internal set-up* those operations that require us to stop algorithm execution, particularly the modification of the actual source code. For example, opening the original source code to declare a new variable or modify a function call would be considered internal set-up. *External set-up* are all those activities that can be done while the program is running. This would include anything that does not modify the original source code.

IV. METHODOLOGY

For this paper, we designed a genetic algorithm implementation using the SMED methodology. However, the same procedure can be used for any other meta-heuristic. We developed three implementations of a genetic algorithm, each one representing a different stage of the SMED process. At each stage, we assume that the set-up time is the time required to go from runt to runt+1 where each run has a different configuration in terms of the functions that it uses. Therefore, the set-up time is the time it takes to take an error-free genetic algorithm, change some of its functions and parameters and run it with its new configuration.

A. Stage 1

Stage 1 requires that we observe the existing method and break it down into its internal and external elements. Our existing method is a canonical genetic algorithm. An example of the pseudo-code of such a genetic algorithm is:

- Initialize algorithm (set state variables)
- Initialize population
- While stopping condition is false
  - Generate children population
  - Replace solutions in current population with children solutions
  - Compare best solution of current population with overall best
  - Update metrics
  - Check stopping condition
- Produce output

This pseudo-code represents what we will call the main function. Each step of this algorithm represents the components that we wish to switch at any given time. The level of detail of the main function (the number of steps) is up to the designer and depends on which specific functionalities s/he could wish to modify in the future. We decided for the current one in order to keep it as a single loop. Following our analogy, each step represents a machine tool that can be changed depending on the configuration of the part going through the process.

In addition to these steps of the algorithm, there is a set of parameters and variables that are required by the program. The variables determine the current state of the algorithm. For example, we need to store the current population, the best solution found so far and its fitness. Parameters are values provided by the user and provide the algorithm with any additional information it needs in order to run. Examples of parameters are the input data file, the stopping condition, the population size, etc.

Now that we have analyzed the structure of our algorithm, we can determine its internal and external elements. In this case, the internal elements are all those things that require the designer to modify the source code. This includes all functions, variable names and variable values that are hard-coded. The only external elements we have at this point are the user provided parameters and the variable values that are set by the algorithm.

This combination of internal and external elements means that, in order to add some functionality to our algorithm, we need to perform the steps shown in Fig. 2.

Each one of these steps requires either modifying at least one line of the main function's source code, or spending time verifying that the changes were done without errors. Even when done without any mistakes, this represents a significant amount of time wasted in order to add a small amount of functionality. The specific implementation of this stage was called ``GA Basic" and consists of a single procedure that executes the steps of the canonical genetic algorithm. Variable names and values are hard-coded. The user inputs a pre-defined number of parameters that the algorithm requires.

B. Stage 2

In Stage 2 we separate internal and external elements. Any change that requires modifying the original source code is considered an internal element. At this point, we have developed a new implementation of the genetic algorithm that we called “GA Modular.” In this implementation, though all the code is contained in a single source code file, the steps of the main function have been separated into different functions. In addition, no variables are hard-coded - their values are either assigned dynamically or are received as user-provided parameters. The parameter list, however, remains pre-defined. Nonetheless, these simple changes reduce the possibility of errors significantly. This coincides with the results of implementing SMED in other domains [1].

A quantitative analysis of this improvement is presented in Section 5. Furthermore, in terms of execution time, it performs as quickly --- and at times faster --- than the original

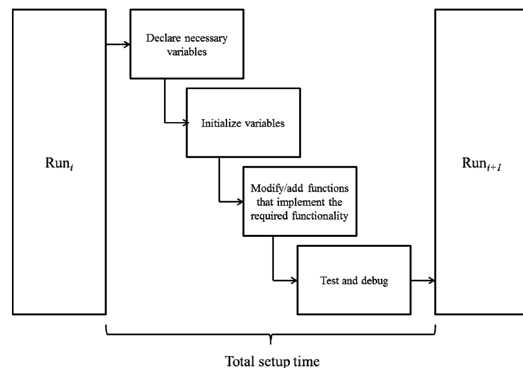

Fig. 2. Steps required to modify GA execution in Stage 1

implementation.

C. Stage 3

Stage 3 requires that we convert internal operations into external operations. This is achieved in the implementation called “GA Improved”. It is composed of several principal components: an algorithm engine, a function database and a function library. The engine is in charge of setting up and controlling the execution of the genetic algorithm. It receives as input the ordered list of functions that the user desires to execute and generates a data structure that will contain all the required state variables and function parameters. The user can also input other kinds of parameters that are necessary to configure a particular run of the genetic algorithm. All this is achieved without the algorithm designer modifying the source code. In order to do this, we require certain information about the functions. This information is stored in the function database, which contains the function names, as well as the variables and parameters that it requires. The database manager is used to access the database and retrieve all this information. Finally, the function library contains the actual source code of each function.

As of now, we have reduced set-up times significantly, however, we have not eliminated them completely. Our implementation only allows us to set-up a different configuration of the algorithm without the need of modifying the source code, that is, we have eliminated the possibility of error during set-up, but not the set-up itself. We are not yet capable of automatically switching from one configuration to the next. Reducing the possibility of set-up errors significantly (see Section 5) comes at a cost. Execution time increases in some runs by a factor of up to 3. Therefore, there is still some room for improvement.

D. Stage 4

Stage 4 consists of improvements on all operations. Some minor changes were done, including the use of alternative data structures for some of the algorithm state variables which resulted in a reduced execution time. The most important improvement, however, was the automation of the set-up procedure. A user can define a set of run configurations in a matrix, feed them into the engine, which will then run the genetic algorithm under all these configurations. This functionality was enabled by storing all the information about each function and its required variables and parameters in the database and then having the engine “build” a specific run with the given configuration.

All of this is done without any changes being done to the main function's source code. The flow of the system is described in Fig. 3. The pre-loop functions are those functions that are executed once before the main GA loop and can be considered the “set-up” functions. The main GA loop contains those functions that execute the GA itself, and, lastly, the post-loop functions include those that store or display the outcome of the current run.

This final version of the implementation is more of a generic meta-heuristic engine than a specific genetic algorithm. If we include the necessary functions in the library to perform some other kind of search, it will run them as such. This is done without modifying any of the main function source code, meaning that the developer can focus on coding the different components and not worry about putting them all together

V. Experimental Results

A. Metrics

To have some measure of the effectiveness of our proposed approach, we had to come up with a quantitative measure of the performance of our solution. Despite the fact that SMED is used for reducing set-up time, we determined that using set-up time as a metric would not be an appropriate measure of improvement due to the difficulty of measuring it, as well as its variability from programmer to programmer. Instead, we focused on the idea of Potential Error Opportunities (PEO). A PEO is defined as the minimum number of potential errors that can be introduced when modifying the source code of the main function. We assume that the more lines of code you have to modify, the longer the set-up time will be. These lines of code represent opportunities to introduce programming errors into an error-free program. Programming errors cause time to be spent debugging and correcting them, which, in turn, increases the set-up time. Therefore, PEOs accurately represent an objective and easy to calculate measure of set-up time.

Three types of programming errors were considered: syntax errors, run-time errors and logic errors. The type of error relates to the set-up time in that syntax errors take the least time to be detected and corrected, while logic errors take the longest. Syntax errors are those where the programmer does not respect the programming language's rules, such as missing a semi-colon at the end of an instruction. Syntax errors are

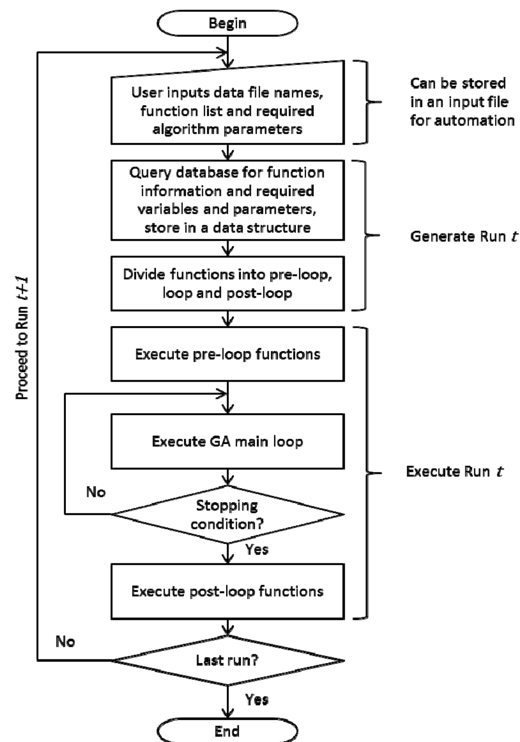

Fig. 3. Meta-heuristic engine execution flow

usually detected before compilation and are often highlighted by the development environment (IDE). Run-time errors are those where the user respects the language syntax but the program execution leads to a prohibited operation (e.g. division by zero or an invalid array index). Run-time errors are only detected during program execution and often cause the program to crash. IDE's usually provide hints as to why the program failed. Logic errors cannot be detected by the IDE and are only noticed when the program does something that was not intended. They often lead to run-time errors and the only way the programmer can find and correct them is by browsing through the source code and following the flow of the program.

As an example, consider changing the initial population generating procedure in our canonical genetic algorithm. Initially, we generate the population randomly and we desire to change this so that the population is obtained from an external file. The following steps would be required to perform this transformation (note that some of the previous code can be re-used so these are only incremental steps needed to add the new functionality):

1. Declare variable for the population data file name
2. Assign population data file name value to the variable
3. Load and read the file
4. Store data in population matrix

This implies that a certain amount of lines of code need to be incorporated into our main function. However, not all lines of code are created equal. Depending on what each line of code does, it can introduce different types of errors. Every line of code that is added can introduce the possibility of a syntax error. Variable value assignments, object instantiation, use of array indices and mathematical operations can lead to run-time errors. Program flow statements (for loops, while loops, etc.) as well as the misplacement of a function call can lead to logic errors.

In order to quantify the PEOs in our example we would proceed as follows: each line of code added would count as a syntax PEO; each variable assignment, function invocation, object instantiation, access to an array index and similar operations would count as a run-time PEO; finally, each program flow statement used would count as a logic PEO.

The second metric we considered was the program's execution time. If the set-up time reduction resulting from a flexible genetic algorithm is outweighed by an extremely high execution time, then it can be argued that such an approach is not worthwhile. Hence, we decided to observe the total execution time in all our implementations. Though we expected it to increase, we wanted to verify that the increase was not such as to render the rest of our effort worthless.

B. Data

We ran our genetic algorithm with four different instances of the Quadratic Assignment Problem (QAP) and one instance of the Travelling Salesman Problem (TSP). The first QAP instance was the 15 locations and 15 facilities problem from [12]. The second instance was a 30 location problem proposed by Krarup and Pruzan known as Kra30a and was obtained from the QAPLib (http://www.opt.math.tu-graz.ac.at/qaplib/inst.html). The other two instances of the QAP were randomly generated with 90 and 150 locations respectively. The TSP we used consisted of 48 cities (state capitals) and forms part of the TSPLib library (the version we used was downloaded from http://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html).

C. Experiments

We performed a series of experiments to test the performance of our solution. Three of these experiments sought to study the improvement in terms of PEOs. Then an experiment was run to test the execution time of the programs, one for each implementation that was developed, for different sized QAPs.

D. Results

1) Potential Error Opportunities

The first focus of our experimentation phase was to analyze the impact of our proposed approach in terms of PEO reduction. We compared the GA Basic, GA Modular and GA Improved implementations. In each experiment, we assumed we began with a canonical genetic algorithm and counted the PEOs introduced into the main function in order to go from canonical to the modified version. In the case of GA Modular and GA Improved, this meant that we did not count the PEOs for developing the function itself because they were not introduced directly into the main function, we simply "connected" them. For GA Basic, we did because those PEOs were introduced into the main function itself. This immediately revealed the benefits of designing algorithms where the internal and external elements are separated.

a) Individual changes to the original code

The first experiment sought to measure the improvement in terms of PEOs obtained when adding individual functionalities. A summary of results is displayed in Table 1.

TABLE I. RESULTS OF EXPERIMENT 1

BASIC			
	Syntax	**Run-time**	**Logic**
Initial solution generation	15	7	2
Parent selection	15	5	3
Crossover	20	12	3
Stopping criterion	7	5	2
Evaluation function	9	1	1
Sub-total	66	30	11
TOTAL	107		
MODULAR			
	Syntax	**Run-time**	**Logic**
Initial solution generation	2	1	1

Parent selection	1	1	1
Crossover	1	1	1
Stopping criterion	5	3	1
Evaluation function	8	1	1
Sub-total	17	7	5
TOTAL	29		
IMPROVED			
	Syntax	**Run-time**	**Logic**
Initial solution generation	0	0	0
Parent selection	0	0	0
Crossover	0	0	0
Stopping criterion	0	0	0
Evaluation function	0	0	0
Sub-total	0	0	0
TOTAL	0		

Our Improved solution completely eliminates the need of modifying the original source code. In SMED terminology, we have met the goal of eliminating the need for set-up. At the same time, the Modular implementation represents already an important improvement over the original, even when it essentially consists of re-organizing the source code.

b) Series of changes to the original code

The second experiment sought to analyze the compounding effect of performing several changes to the original algorithm configuration, as one would if experimenting with various configurations. We performed five basic modifications and their combinations to test the effects of a variety of configurations, these five modifications are:

1. Change initial population generation procedure from random to file based
2. Change parent selection method from random to tournament based
3. Change crossover process from a single-point crossover to a uniform crossover
4. Change population replacement strategy from a pooled best to one preserving all mutants
5. Change stopping criterion from time-based to number of generations

Now, if a researcher decides to test all the combinations of these five modifications in a full-factorial design, s/he would require executing at least 32 runs, each with a different configuration. In such a scenario, the number of PEOs would grow very rapidly. An analysis of such scenario revealed that GA Basic would introduce 1712 PEOs (1024 syntax, 512 run-time and 176 logic PEOs), GA Modular 352 (160, 112 and 80 respectively) and GA Improved would remain at 0.

For this paper, we chose at random 6 of those 32 runs, as well as a “worse-case” scenario where all the modifications are added. A run can be described as an array where 'Yes' means that modification i = 1,…,5 is done, and 'No' means that it is not used. The characteristics of each one of these runs are shown in the following list:

1. Run 1: Yes,No,Yes,No,Yes,Yes,No
2. Run 2: Yes,No,No,Yes,No,No,Yes
3. Run 3: Yes,Yes,No,No,No,No,No
4. Run 4: No,Yes,No,No,No,No,Yes
5. Run 5: Yes,No,Yes,Yes,No,Yes,Yes
6. Run 6: Yes,Yes,Yes,Yes,Yes,Yes,Yes

The results of these runs are presented in Table 2.

TABLE II. RESULTS OF EXPERIMENT 2

BASIC			
Run	**Syntax**	**Run-time**	**Logic**
1	7	5	2
2	27	17	5
3	34	20	6
4	22	10	5
5	37	17	7
6	49	27	8
7	64	32	11
Sub-total	240	128	44
TOTAL	412		
MODULAR			
	Syntax	**Run-time**	**Logic**
1	5	3	1
2	6	4	2
3	7	5	2
4	6	4	2
5	8	5	3
6	9	6	4
7	10	7	5
Sub-total	51	34	19
TOTAL	104		
IMPROVED			
	Syntax	**Run-time**	**Logic**
1	0	0	0

2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
Sub-total	0	0	0
TOTAL	0		

These results are only for a full-factorial with five factors at 2 levels each. In a real implementation of a meta-heuristic algorithm, it is common to have more than two levels for each factor, which would drive the number of possible configurations up, and with it, the number of PEOs. Therefore, we believe our approach offers a huge opportunity for more vast experimentation without wasting time in performing changes to the original source code.

c) Changing the problem

Our third experiment sought to study the effect of changing the problem to solve. We restricted ourselves to a simple switch from a QAP to a TSP. Both problems are fairly similar and allow for the re-use of significant portions of their code. Even so, GA Basic required a total of 44 PEOs, GA Modular only required 3 and GA Improved 0.

2) Execution Time

The second experimental phase focused on the program's execution times. We observed during our trial runs that there was a noticeable increase in execution time when we used GA Improved. This, of course, was expected because of the overhead that allows us to dynamically configure the runs but we were concerned with this becoming a serious issue as we dealt with more realistic problems. Therefore, we considered two variables that could drive execution time: the number of iterations and the problem size. We ran the different implementations with different sizes of the QAP (15, 30, 90 and 150) and with different number of generations, ranging from 10 to 10,000.

Fig. 4 shows the change in execution time as the number of iterations increased for the QAP 15.

It is clear from the graph that something within each iteration is causing the increase in execution time for GA Improved since its increase rate is much larger than that of the alternative implementations. The procedure used for dynamically calling a function is, as should be expected, slower than the traditional, direct call. This is a cost we must pay for flexibility. But it still remained to verify whether this would become a roadblock as we dealt with larger problems. Much to our surprise, our tests with larger problems revealed that GA Improved was closing the gap in terms of execution times. Finally, for the QAP 150, it ran faster than both GA Basic and GA Modular. Fig. 5 shows the outcome of our experiment.

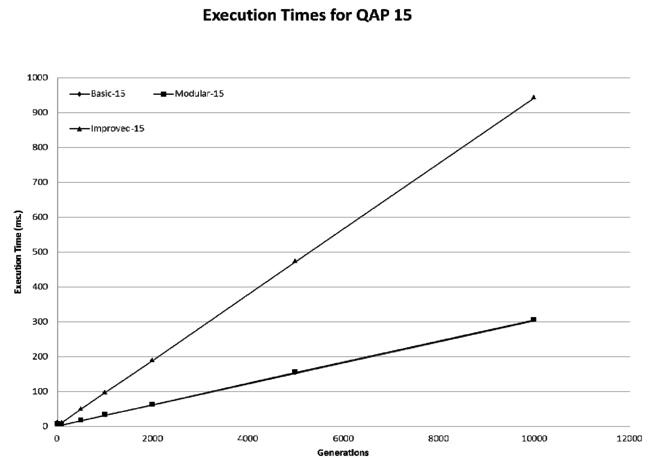

Fig. 4. Execution times for QAP 15

We believe that the use of different data structures in GA Improved (Hash Maps and Vector objects) for our state variables allowed a quicker access to the data, as opposed to the more primitive arrays and matrices used in GA Basic and GA Modular. As the data structures grew in size, the optimized access functions to the data made up for the initial overhead, leading to GA Improved's faster execution.

TABLE III. AVERAGE EXECUTION TIMES FOR DIFFERENT SIZES OF THE QAP (TIME IN MILLISECONDS)

Size	Generations	Basic	Modular	Improved
15	10	9.5	5.8	4.7
	100	4.5	4.2	11.2
	500	16.1	16.1	49.8
	1000	31.5	31.8	96.8
	2000	61.1	61.8	189
	5000	152	154.5	472.6
	10000	302.8	305.2	942.2
30	10	2.6	2.3	2
	100	12.4	12.4	17.7
	500	56.2	59	88.9
	1000	111.9	113.2	173.7
	2000	221.3	225	348
	5000	552.2	561.6	866.2
	10000	1110.6	1118.6	1729.4
90	10	12.2	12.4	10.3
	100	98.6	99.2	98.2
	500	482.6	489.6	489.8
	1000	978	988.6	981.4
	2000	1927.1	1938.8	1963.7
	5000	4809.8	4840	4907.1
	10000	9651.9	9692.8	9853.4

150	10	38.3	37.4	27.3
	100	275.4	276.7	265.4
	500	1355.6	1368.2	1326.8
	1000	2717.5	2734.6	2621
	2000	5432.8	5443.2	5206.6
	5000	13481.1	13646.7	13120.5
	10000	26985	27289.8	26104.5

VI. CONCLUSIONS AND FUTURE WORK

We successfully performed a set-up reduction procedure for the design of a genetic algorithm, showing that a well-thought design can significantly improve the experimental process. We demonstrated the use of this methodology by developing a genetic algorithm and analyzing two metrics: the Potential Error Opportunities (PEOs) and total algorithm execution time. We compared our proposed approach to two other implementations that we had developed previously and demonstrated its superiority in terms of PEO elimination. Furthermore, we found that execution time improved as the problem size grew proving that our solution is very satisfactory.

An approach like the one we proposed is beneficial for researchers that work in the area of meta-heuristics in general, as it provides them with a framework for designing algorithm implementations that are flexible and easily tested. Too much effort and time are invested in experimenting with a variety of configurations of the algorithm. Though this experimentation is necessary, the amount of time and effort used is not. It is possible to design programs that allow us to do this quickly and efficiently, with a minimum amount of programming errors. This enables a researcher to experiment with a wider variety of configurations, possibly many which he would have otherwise left untested.

This work can be further developed in many directions. One is that of the automation of the testing of different configurations. Our implementation allows this to a degree. It is possible to test a variety of configurations, and print out their output without him having to modify the genetic algorithm's source code. It would be desirable to include more user-friendly and automated ways of doing this. This will significantly reduce the time that is invested in testing alternate configurations and reduce the possibility of other errors. Furthermore, such automation could enable the development of adaptive searches that modify their search procedure (that is, their structure) automatically in response to the feedback they receive. This allows for the exploration of vast search spaces without the researcher having to perform changes manually. The development of an external function library opens the door for extensive collaboration. Specific functions can be sent from one researcher to another and easily plugged into their algorithms. Libraries can also be stored in servers or multiple machines and allow the development of distributed meta-heuristics that perform their in parallel. A next step could be the development of a graphic user interface for maintaining the function library, as well as one for configuring runs.

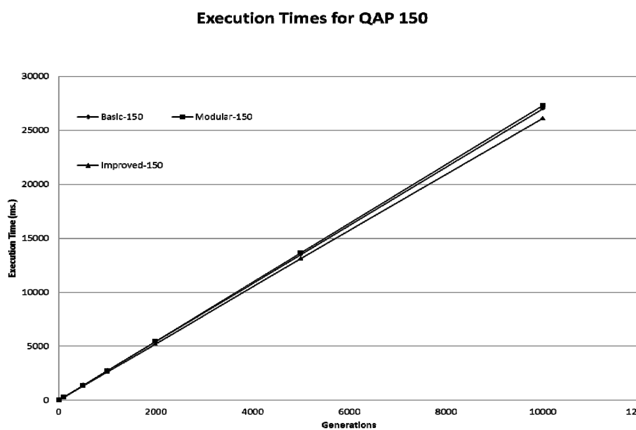

Fig. 5. Execution times for QAP 150

REFERENCES

[1] J.T. Black and S.L. Hunter. Lean Manufacturing Systems and Cell Design. McGraw-Hill, 2003.

[2] S. Cahon, N. Melab, and E.-G. Talbi. Building with ParadisEO reusable parallel and distributed evolutionary algorithms. Parallel Computing, 30(5/6):677 - 697, 2004.

[3] L. Di Gaspero and A. Schaerf. Easylocal++: An object-oriented framework for the design of local search algorithms and meta-heuristics. In MIC2001 4th Meta-heuristics International Conference, pages 287-292. Citeseer, 2001.

[4] M. Gilmore and D.J. Smith. Set-up reduction in pharmaceutical manufacturing: an action research study. International Journal of Operations & Production Management, 16(3):4-17, 1996.

[5] D. Van Goubergen and H. Van Landeghem. Rules for integrating fast changeover capabilities into new equipment design. Robotics and Computer-Integrated Manufacturing, 18(34):205-214, 2002.

[6] B. J. Hicks. Lean information management: Understanding and eliminating waste. International Journal of Information Management, 27(4):233-249, 200`7.

[7] M. Lukasiewycz, M. Gla, F. Reimann, and J. Teich. Opt4J: a modular framework for meta-heuristic optimization. In Proceedings of the 13th annual conference on Genetic and evolutionary computation, GECCO '11, pages 1723-1730, New York, NY, USA, 2011. ACM.

[8] R. McIntosh, S. Culley, G. Gest, T. Mileham, and G. Owen. An assessment of the role of design in the improvement of changeover performance. International Journal of Operations & Production Management, 16(9):5-22, 1996.

[9] R. McIntosh, G. Owen, S. Culley, and T. Mileham. Changeover improvement: Reinterpreting Shingo's "SMED" methodology. IEEE Transactions on Engineering Management, 54(1):98, 2007.

[10] T. Melton. The benefits of lean manufacturing. Chemical Engineering Research and Design, 83(6):662{673, 2005.

[11] C. Moxham and R. Greatbanks. Prerequisites for the implementation of the SMED methodology: A study in a textile processing environment. International Journal of Quality & Reliability Management, 18(4):404-414, 2001.

[12] T.E.Rum J. Nugent, C.E.Volhnann. An experimental comparison of techniques for the assignment of facilities to locations. Operations Research, 16(1):150, 1968.

[13] S. Shingo. A Revolution in Manufacturing: The SMED System. Productivity Press, Cambridge, MA, 1985.

[14] D. Trietsch. Some notes on the application of single minute exchange of die (SMED). Technical report, DTIC Document, 1992.

[15] S. Coble Trovinger and R.E. Bohn. Setup time reduction for electronics assembly: Combining simple (SMED) and IT-Based methods. Production & Operations Management, 14(2):205-217, 2005.

[16] J.P. Womack. The Machine that Changed the World. Free Press paperbacks. Scribner, 1990.