Automated Capacity Planning for PEPA Models

Automated Capacity Planning for PEPA Models

Christopher D Williams and Jane Hillston

LFCS, School of Informatics, University of Edinburgh
jane.hillston@ed.ac.uk
http://www.quanticol.eu

Abstract. Capacity planning is concerned with the provisioning of systems in order to ensure that they meet the demand or performance requirements of users. Currently for PEPA models, a modeller who wishes to solve a capacity planning problem has to either carry out a manual search for an optimal configuration or work outside the provided tool suite. We present a new extension to the Eclipse Plug-in for PEPA which integrates automated capacity planning into the functionality of the tool, thus allowing optimal configurations of large scale PEPA models to be found.

1 Introduction

Performance analysis occurs in many guises during system development. One important role of a performance analyst is as a capacity planner, helping the system developers to appropriately dimension their system in order to meet user demand in a satisfactory manner. This involves choosing the appropriate number of copies for each type of component within the system, for both software components (e.g. threads) and hardware components (e.g. database servers). This can involve running models of the system with many different configurations to thoroughly explore the parameter space to find an optimum. There is often a tension between the number of components and the user perceived performance, as system managers wish to limit the number of components for a variety of reasons, including economic, efficiency and maintainability considerations.

The Eclipse PEPA Plug-in is a mature analysis tool supporting the PEPA modelling language, and offering a number of different solution techniques. Whilst it does offer an experimentation facility, this is intended for varying one or two parameters within a model and all results are returned to the modeller. In contrast, in capacity planning the modeller will typically want to thoroughly explore a multi-dimensional parameter space but only be presented with results for those points in parameter space which are optimal with respect to some modeller-defined performance and population target. Currently a modeller who wishes to conduct such an exploration must manually search parameter space, or export their model to another format such as Matlab and then code up a search algorithm themselves.

In this paper we present an extension of the Eclipse PEPA Plug-in which incorporates an automated capacity planning tool which addresses this problem.

The modeller can specify their optimisation problem and let the tool search for a model configuration which best matches the performance target whilst also keeping the number of components minimal. In the current implementation this is aimed at scalable PEPA models amenable to solution based on ordinary differential equations, making interactive capacity planning possible. But the developed framework is flexible and now that the feasibility of the approach has been established, could be readily extended to a broader class of PEPA models and other solution techniques.

1.1 Related Work

Hillston *et al.* suggested the feasibility of a capacity planning tool for PEPA in [1]. Within that paper, the parameter space of a moderately-sized example is explored by hand to find a configuration which matches a target average response time, when one population size is fixed and all others are allowed to vary. This paper provided inspiration for our current work. Genetic algorithms and genetic programming metaheuristics have previously been used in conjunction with PEPA and Bio-PEPA models by Marco *et al.* [2, 3]. In that work they sought to find model parameters which give optimal fit of model output to a given time series of biological data. Both activity rates and model structure make up the search space for the metaheuristic. Similarly Karaman *et al.* use genetic algorithms to construct a process algebra model satisfying a path optimisation property, again focussing on the time series view of the process algebra model output [4]. In contrast our work identifies emergent global properties of the process algebra model as the goal. Like Karaman *et al.* our primary focus is on investigating model with different structures, i.e. different numbers of components in this case. The work of Geisweiller also sought to match to given performance characteristics but by finding optimal rate parameters for a PEPA model with fixed structure, using expectation-maximisation techniques [5]. More generally, in [6], Cerotti *et al.* present a general capacity planning tool for dimensioning in Cloud systems, based on simple queueing abstractions.

1.2 Structure of the Paper

The rest of this paper is organised as follows. In Section 2, we present the necessary background. In Section 3 we describe the basic functionality of the tool and how a user can specify a search based on a performance target and cost requirements. The search procedure is sensitive to the settings of the algorithm, so we additionally offer a *driven* search in which another, simpler, metaheuristic, is used to find the best settings for the intended search. This is explained in detail in Section 4. In the following section, Section 5 we present some results from the tool, run on a number of different models, and in particular show how the Particle Swarm Optimisation (PSO) search compares with simpler metaheuristics such as hill-climbing and a genetic algorithm. In Section 6, the paper concludes with a summary of the results and a discussion of how the work can be developed further.

2 Background

In this section we give a brief overview of the background to the project, reviewing the PEPA modelling language, and specifically the fluid approximation which allows large scale models to be rapidly solved via a set of ordinary differential equations. In this work we choose this solution technique to underlie our capacity planning because we envisage a tool which the user will engage and experiment with. However, the same framework could be used with alternative solution methods albeit with slower response time and the capacity planning would no longer be interactive. Finally in this section we describe the meta-heuristic that is the principal focus of our work.

2.1 PEPA

PEPA is a CSP-like process calculus extended with the notion of exponentially distributed activities [7]. A PEPA model consists of a collection of *components* or *processes*, which undertake actions. A component may perform an action autonomously, *independent actions*, or in synchronisation with other components, *shared actions*. PEPA models are generated by the following two-level grammar:

$$S ::= (\alpha, r).S \quad | \quad S + S \quad | \quad A_S, A_S \stackrel{def}{=} S$$
$$C ::= S \quad | \quad C \underset{L}{\bowtie} C \quad | \quad C/L \quad | \quad A_C, A_C \stackrel{def}{=} C$$

The first production defines *sequential components*, i.e., processes which only exhibit sequential or branching behaviour (by means of prefix, ".", or choice, "+", respectively). The second production defines *model components*, in which the interactions between the sequential components are expressed through the cooperation ("$\underset{L}{\bowtie}$") and hiding ("/") operators. Within a cooperation, the set L specifies which action types must be shared; components can proceed independently and concurrently on other action types. A *system equation* specifies all the components within a system and how they must interact.

Typically, each sequential component corresponds to a component of the system and the performance of the system is constrained by the interactions between components as imposed by the cooperations. For example for a client-server system, some number of clients may compete for access to a limited number of servers. This may be written as the system equation

$$Client[N_c] \underset{\{request\}}{\bowtie} Server[N_s]$$

where $Client[N_c]$ is shorthand for $Client \underset{\emptyset}{\bowtie} \cdots \underset{\emptyset}{\bowtie} Client$ for a population of N_c clients, and similarly for $Server[N_s]$.

The capacity planning problem is to find appropriate population sizes for the components in the system equation which allow the system to meet a performance target. For example this might be *response time should be on average less than 2s when there are 100 clients in the system.* Some populations, such as the clients, may be fixed as they are specified by the requirement, whereas others

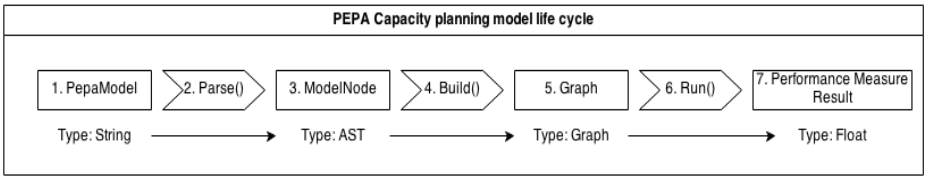


Fig. 1: The model life cycle in the capacity planning tool. The square boxes represent classes of models, the arrowed boxes represent a conversion method.

may vary, allowing the modeller to explore a parameter space. In this simple case, the parameter space is one-dimensional and capacity planning amounts to finding the number of servers which is sufficient to meet the response time target. However, in general the search space will be multi-dimensional as the system will be made up of many different interacting populations of components.

The original structured operational semantics for PEPA [7], gives rise to a continuous time Markov chain (CTMC) via a labelled multi-transition system. In [8], an alternative symbolic semantics in terms of generating functions is presented, which allows a fluid approximation of large scale PEPA models to be derived automatically. This derivation is incorporated in the tool which supports PEPA modelling, the PEPA Eclipse Plug-in tool [9]. Thus the PEPA Eclipse Plug-in supports numerical solution of the CTMC, stochastic simulation and ODE numerical simulation.

Our capacity planning currently focuses on this latter approach: the scalability and efficiency of the ODE-based fluid approximation means that many model instances can be solved relatively quickly, providing an interactive experience for the user. Moreover, the large scale models amenable to this approach are typically those for which it is difficult to predict the relative influence of one individual over all the interacting populations.

To evaluate a PEPA Model using ODEs, the PEPA Eclipse Plug-in first converts the String model class (`PepaModel` — the model as input by the modeller) into a Graph model class (`Graph`), via an abstract syntax tree termed the `ModelNode`. Once the model is a graph, the ODEs can be evaluated, returning performance measure results as an array of floats. Fig. 1 shows the PEPA Eclipse Plug-in methods used by the capacity planning tool, and the life cycle of a model.

The intervention of the capacity planning tool, compared with a regular ODE solution of a PEPA model, is that the capacity planning tool manipulates model configurations during step 3 of the lifecycle. Using a Visitor pattern and a Java class called `ASTHandling`, the capacity planning tool can operate on models, and change the population value of one or more components. This updated AST is then built into a `Graph` (step 5) and evaluated using ODEs. This will be explained in more detail in Section 3.

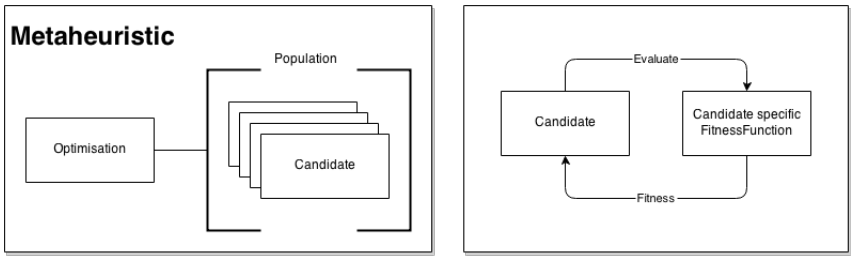


Fig. 2: Schematic view of a metaheuristic

2.2 Particle Swarm Optimization (PSO)

Metaheuristics are a class of general algorithms which may be used to find optimised solutions to a large class of problems. Examples include hill-climbing, genetic algorithms, simulated annealing, ant colony optimisation and particle swarm optimisation (PSO). These can be thought of as strategies for guiding a search, and they do not typically guarantee convergence or optimality. But in many practical situations they have been found to perform well, both with respect to optimality and efficiency compared with brute-force or manual search. During the course of developing the capacity planning tool we implemented and experimented with a number of different metaheuristics: hill-climbing, a genetic algorithm and PSO. The experimentation suggested that the PSO was the most successful in the sense of both efficiency and optimality. Thus in the final version of the tool this is the offered algorithm, and in this paper we focus on that due to space limitations, although we will present some of the experimental results in Section 5.

Initiation of variables

```
generation = user defined value            //number of iterations
candidatePopulation = user defined value   //how many candidates
localBest = user defined value          //weight of local best in new velocity
globalBest = user defined value         //weight of global best in new velocity
originalVelocity = user defined value  //weight of original velocity in new velocity
```

Scattering of candidates

```
bestCandidate = null
arrayOfCandidates = []

for candidatePopulation do:
    newCandidate = (new candidate) //candidate random position and velocity
    arrayOfCandidates = arrayOfCandidates ∪ newCandidate
```

Fig. 3: PSO initiation

Iterative search

```
for generation do:

   //find fittest
   for each candidate in arrayOfCandidates do:
      fitnessFunction(candidate) //use system equation fitness function to assess fitness
      if(candidate 'is better than' bestCandidate) do:
         bestCandidate = candidate

   //update each candidate
   for each candidate in arrayOfCandidates do:
      currentPosition ← candidate's position // the system equation
      previousVelocity ← candidate's velocity
      localBestPosition ← candidate's previous best position
      globalBestPosition ← bestCandidate's best position
      newVelocity = (originalVelocity * previousVelocity) +
         localBest * (localBestPosition - currentPosition) +
         globalBest * (globalBestPosition - currentPosition)
      candidate's velocity ← newVelocity  //the candidate gets a new velocity vector

   //move each candidate
   for each candidate in arrayOfCandidates do:
      candidate's current position ←
         floor(candidate's current position + candidate's velocity)
```

Fig. 4: PSO search

PSO is a stochastic optimisation algorithm modelled after flocking or swarming agents [10]. A PSO works by scattering a number of candidates (or particles) in some search space and providing each with a random velocity. Each generation, or iteration, of the optimisation method, each candidate moves according to its velocity. Then the best candidate of that iteration, called the *global best*, is found and its position is made known to all other candidates. Each agent then uses this global best, its own velocity, and its own best position historically, to create a new velocity vector which it uses in the next step. After a number of iterations the PSO should converge on an optimum position in the search space. (See Figs. 3 and 4 for pseudocode inspired by Luke [11].)

3 Simple Search

In the basic use-case for our tool, the modeller uses the capacity planning tool to set up a search directly. In this case the modeller establishes a fitness function with the aid of wizard in the Eclipse Plug-in. As will be explained later in this section, the fitness function is constructed of a number of components, allowing a target performance measure, and the population sizes to be taken into consid-

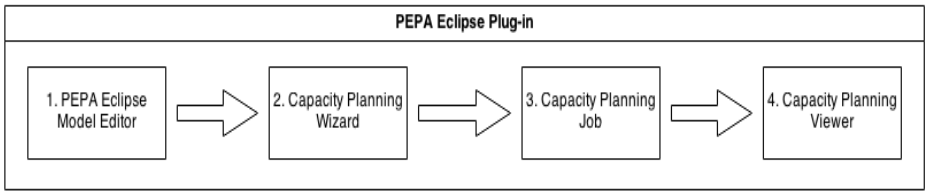


Fig. 5: Steps in the use of the capacity planning tool. Input is a PEPA model from the Editor, the output is displayed in a Viewer pane within Eclipse.

eration. The wizard also allows the modeller to choose the setting for the PSO algorithm, such as how much weight should be given to the global best position in the definition of a new velocity for each candidate. In this simple search each candidate in the PSO corresponds to a system equation, or configuration of the system. Once the search is fully specified, one run of the PSO algorithm is used to explore parameter space and return the candidate which gives the best value of the fitness function. The parameter space is defined by the range of populations considered for each of the component types in the PEPA model.

The steps in the use of the capacity planning tool are depicted in Fig. 5. We will focus on steps 2, 3 and 4.

3.1 Capacity Planning Wizard

An Eclipse wizard is provided to support the modeller in the set up and initiation of a capacity planning job. A `Wizard` is a Java class and is used to present a logical ordering of more Java classes called `WizardPages`. `WizardPages` are used to guide the user in entering the parameters, the input and settings, required for a capacity planning search. Fig. 6 shows the steps of the capacity planning wizard pages.

1. Input is a model created in the PEPA editor.
2. The user starts the capacity planning tool by selecting the appropriate action in the PEPA menu: the wizard picks up the PEPA model from the editor. Each of the following steps corresponds to a page in the wizard.
3. The user sets the type of search, driven or single (explained in the following section), and the kind of performance target they seek to address: currently either response time or throughput.
4. Driven or single search are specified separately. Here the user can change the number of experiments (explained in the following section) and algorithm settings.
5. This page is for the specification of the performance targets, selecting actions in the case of throughput, and agent states for response time. Settings for the ODE solution function are also selected.
6. This and the following pages specify the form of the fitness function. Here the relative weights of the population and performance are set.

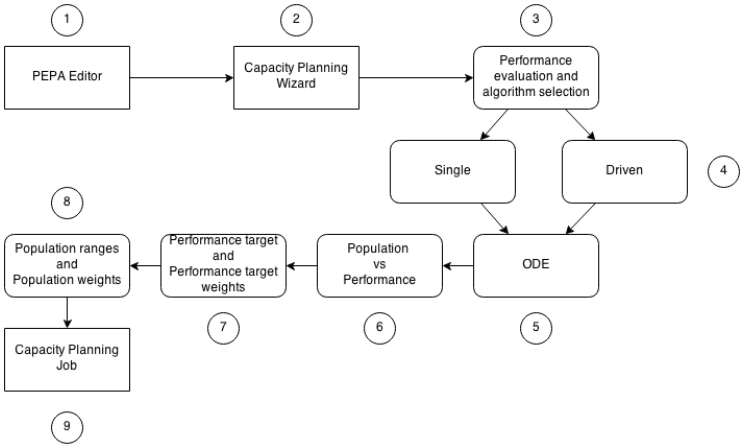


Fig. 6: A high level picture of the capacity planning wizard.

7. The modeller specifies the performance target values, and the relative weighting of the performance targets, if there is more than one.
8. Here the modeller sets the possible population range for each component type in the model and so determines the parameter space for the search. Weighting can be assigned for each component type, allowing higher populations for some components to be penalised more than for others.
9. Completion of the previous page finishes the specification of the capacity planning job. Data is passed to the solution engine and search is initiated.

3.2 Capacity planning Job

General design The capacity planning wizard passes all data to the capacity planning Job, and then starts the search. A capacity planning job is created as a separate thread and so runs separately from the Eclipse user interface. Fig. 7 presents a schematic view of a capacity planning job. As the metaheuristics which we have considered are stochastic, in order to increase the likelihood of finding a good result, a number of searches are run serially; each run is termed an *experiment*, and each set of experiments is termed a *Lab*. Each experiment is a run of the metaheuristic with a randomly seeded *candidate population*.

Each candidate corresponds to a point in the parameter space, i.e. an instantiation of the system equation for the model, and each will have a corresponding value of the fitness function. Since the fitness function has an element corresponding to the performance target, each model instance must be solved. Currently the existing ODE solver within the PEPA Eclipse Plug-in is used for this, but the architecture has been designed so that a metaheuristic can run on any type of candidate, and with any of the solvers present in the tool.

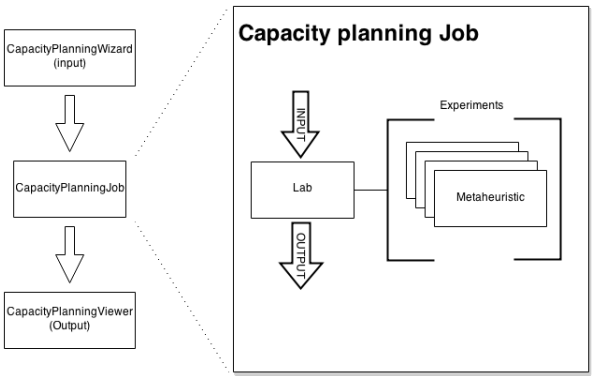


Fig. 7: Schematic view of a capacity planning job

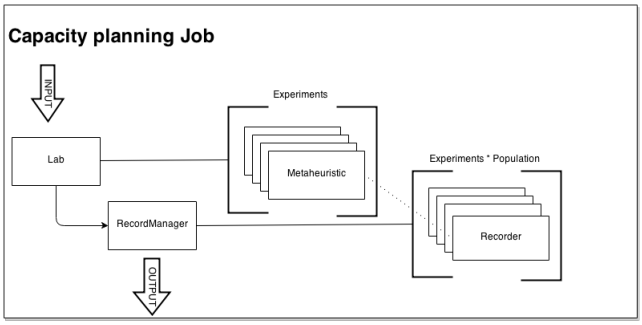


Fig. 8: Recording output

Fitness Function Evaluation The fitness function, built by the modeller with the aid of the wizard, determines the search that is carried out. The PSO seeks to minimise the value of the fitness function, by seeking candidates which have the following characteristics:

- Good performance with regards to the target. This will be based on the result of the performance measures from the ODE function.
- Minimal total component population (*componentPopulation*), i.e. the number of components in the system equation.
- There will typically be a trade-off between performance and population so the modeller gives a relative weight to these objectives (*performanceWeight* vs. *populationWeight*).
- Component weighting ($componentPopulation_i$): similarly within a system, it might be more important to minimise the population size of some component types than others, for example due to cost or other considerations.
- Performance target weighting ($performanceTargetWeight_i$): when the modeller has specified a performance target with multiple elements they can give relative weights to those elements.

These five elements collectively build up the fitness value, which determines how good one system equation candidate is relative to another[1]. The fitness is better the *lower* it is. Thus for each iteration of the PSO we construct:

1. The result of the performance evaluations from the ODE function

The performance results, $ODEResult_i$, is returned by the ODE solver after it has evaluated a model instance. There will be one $ODEResult_i$ for each user defined performance target, $performanceTarget_i$, for each performance measure specified by the modeller in the capacity planning wizard. Any result that is better than the target is given a value of 0. A $scaledPerformanceValue$ is created for each target so that it can be used later in the fitness function:

$$scaledPerformanceValue_i = |(100 - (ODEResult_i / performanceTarget_i) \times 100)|$$

2. The number of components in a system equation

In the wizard the modeller defines the minimum and maximum population for each component type in the system equation: $minPopulation_i$ and $maxPopulation_i$, and from these we derive the range: $populationRange_i$. Each system equation candidate has a population for each component $componentPopulation_i$. Then a value, $scaledPopulationValue_i$, is created for each population:

$$scaledPopulationValue_i = |((componentPopulation_i / populationRange_i) \times 100)|$$

3. The weighting of components in terms of population

The modeller also specifies the weight for each component, $componentWeight_i$, which gives the fitness function some notion of cost per component. From these values we derive the total population weight ($totalCWeight = \sum_0^n (componentWeight_i)$) and the contribution of the population of this candidate to the fitness function:

$$weightedPopResult = \sum_0^n (scaledPopulationValue_i \times (\frac{componentWeight_i}{totalCWeight}))$$

4. The weighting of performance targets across all selected performance targets

Similarly we construct the contribution of the performance target to the fitness function by defining $totalPWeight = \sum_0^n (performanceTargetWeight_i)$ and $weightedPerfResult$:

$$weightedPerfResult = \sum_0^n (scaledPerformanceValue_i \times \frac{performanceTargetWeight_i}{totalPWeight})$$

This allows the user to put more importance on finding one performance target over any others.

[1] Here we use a weighted sum, but of course, a weighted product could also be used.

5. The balance between performance targets and population size

Finally we use the weights for population and performance entered by the modeller, $populationWeight$ and $performanceWeight$ to combine these values in the final fitness value. Here $totalWeight = populationWeight + performanceWeight$

$$\begin{aligned} fitnessValue = weightedPopResult \times (populationWeight/totalWeight) \\ + weightPerfResult \times (performanceWeight/totalWeight) \end{aligned}$$

In summary, this creates a fitness value such that the smaller the value, the lower the number of components used, and the closer the performance evaluation will be to the user defined target.

3.3 Capacity planning Viewer

The capacity planning viewer is a viewer pane in the Eclipse environment. At the end of each run of the metaheuristic it is passed the ten top candidates by fitness value and displays them in order in the view pane.

To keep track of all the experiments and candidates we use a Java class `RecorderManager`. A `RecorderManager` is created with every Lab, and every experiment has a `Recorder`. It is the `Recorder`'s role to track the progress of an experiment, and it is the task of the `RecorderManager` to collect all `Recorders` and pass the results to the capacity planning `Viewer`.

4 Driven Search

As will be discussed in the next section the simple search proved effective for finding a model configuration satisfying the performance targets, whilst minimising the number of components. But in testing users had difficulties in choosing the best settings for the PSO algorithm to achieve the best results. These settings include the initial population and velocity of components, weightings for global and local best as well as the number of generations and experiments. Therefore we experimented with using another metaheuristic to find the settings for a metaheuristic search over the parameter space. We term this *driven search*: a *driving* metaheuristic is used to find the best settings for the second *driven* metaheuristic. After some investigation we found that this works well when a hill-climbing algorithm is used to find the best settings for a PSO algorithm.

In a driven search the candidate is itself a Lab, as defined for single search; this is termed a `Lab candidate`. Each experiment consists of a single search `Lab`, and so one driven experiment, consists of many Lab experiments. This is represented schematically in Fig. 9.

Lab Fitness Function In order to evaluate the fitness of a Lab candidate, we construct a Lab fitness function. This Lab fitness function calls the `RecordManager` from the underlying single search (Fig. 10) to return four values;

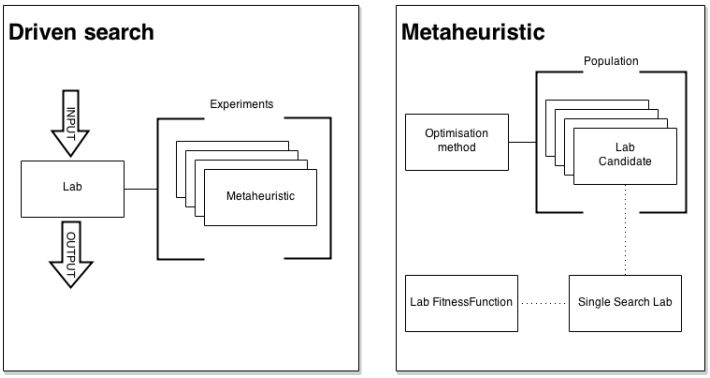


Fig. 9: Schematic view of a driven search

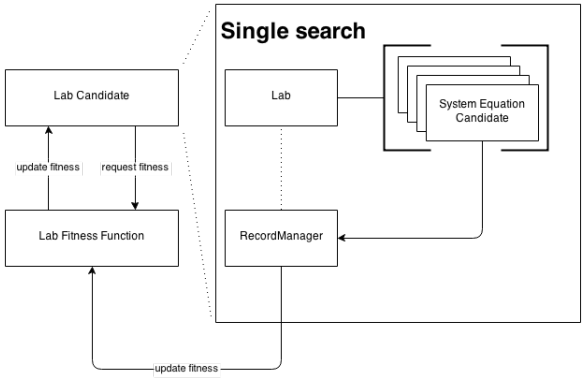


Fig. 10: Lab candidate calling a fitness update

- Top fitness from the underlying experiments (*topFitness*);
- Mean fitness of all experiments in this Lab candidate (*meanFitness*);
- Standard deviation of the best values found for this Lab candidate (*standardDeviation*);
- Average response time of the underlying experiments (*averageResponseTime*)[2].

The lower a lab fitness value is, the better the underlying single search will be at finding the optimal candidate. Getting the best fitness has the highest priority and is reflected in the fitness function by having a weight of 0.6, next priority is finding a single search that on average returns a high value and therefore it uses a weight of 0.2. In order to break any ties between Lab candidates, accuracy (standard deviation) and response time each are given weights of 0.1. These

[2] How long an experiment took to run, not to be confused with the average response time performance measure evaluated on the model.

weights were arrived at through experimentation.

$$\begin{aligned} fitnessValue = (0.6 \times topFitness) + (0.2 \times meanFitness) \\ + (0.1 \times standardDeviation) + (0.1 \times averageResponseTime) \end{aligned}$$

5 Evaluation

As mentioned previously, our initial implementation offered three different meta-heuristics. In addition to PSO we implemented a stochastic hill climbing (HC) and a genetic algorithm (GA). Table 1 shows the results of experimentation with these different search algorithms over a variety of models[3]. The model sizes ranged from 2 populations (Simple) to 9 populations (E University), with components from 2 local states (Traffic) up to 18 local states (E University).

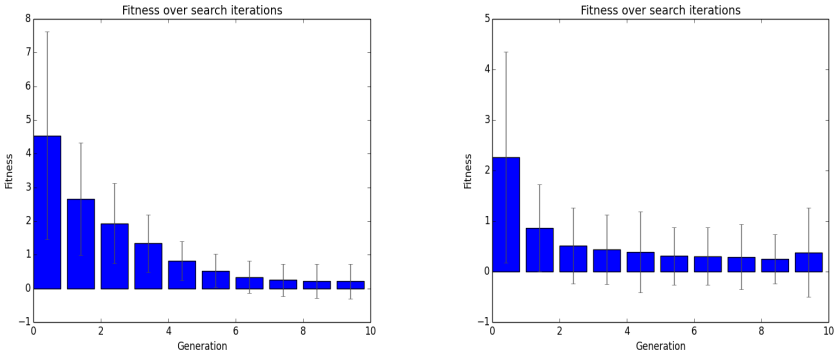
	GA		**HC**		**PSO**	
Model	**Top fitness**	**RT**	**Top fitness**	**RT**	**Top fitness**	**RT**
Brewery	**21.1459**	31.49	23.7710	46.15	22.6990	**29.075**
Brewery2	18.8109	220.39	16.3775	217.16	**14.1821**	**208.05**
E University	11.8022	1723.17	10.5644	**1327.02**	**5.22160**	1653.36
Example System	8.26005	46.14	7.33108	46.75	**3.91710**	**43.98**
Example System2	6.58843	75.75	7.65840	75.67	**1.75361**	**71.2**
Large-t	**2**	120.71	**2**	122.68	**2**	**119.95**
Simple	8.00494	**29.17**	7.50368	33.02	**7.25008**	30.10
Simple2	9.03451	33.82	11.0040	28.45	**4.86591**	**26.70**
Traffic	34.2665	33.05	34.4215	33.32	**32.9015**	**30.87**

Table 1: Evaluation results for Top fitness (to 6 s.f.) and Response time in milliseconds (RT). Best values highlighted in each row.

It can be seen from Table 1 that the PSO algorithm is generally achieving the best result and often in the shortest time. This is the reason why we decided to only include PSO in the final implementation of the tool.

Figs. 11(a) and 11(b) show what happens to fitness over generations with PSO and GA. Each algorithm was run 1000 times on the same model and the average fitness of each generation was calculated. An effective algorithm should improve (decrease) the average fitness through the generations. Figs. 11(a) shows the average fitness and the variance of fitness over generations of the PSO on our example model, `Example.pepa`. On average the PSO has converged after 8 generations — there is no significant improvement in average fitness after that. In contrast, GA convergence appears to happen around the 5th generation, but there is much wider variance, indicating that there are better candidates found

[3] More details of this evaluation and the models used can be found in [12].



(a) PSO (b) GA

Fig. 11: Graph showing the average fitness and variance of fitness over generations of PSO (left) and GA (right).

but not consistently. The width of the variance shows GA has different behaviour on each run, whereas PSO has much less variance in the final generations. The results for HC exhibited even greater variability and much less convergence. These results show that the PSO has the same behaviour on average independently of how the algorithm was started. Thus it is more predictable.

Finally in Table 2 we show the results of a comparison of a single PSO search, against a driven PSO search on each of our example models. Note that the driven search achieves a better fitness result in the majority of models.

6 Conclusions

In this paper we have presented the new capacity planning facility within the Eclipse Plug-in for PEPA. This tool offers both a fast *single* search and a slower *driven* search. The former requires the modeller to understands the heuristic

Model	**Driven fitness**	**Single fitness**
Simple ab	**5.00523**	6.30427
Simple a	**7.25008**	7.25008
Brewery ab	18.6424	**15.1921**
Brewery a	**22.3579**	23.1876
E University	**5.11754**	6.06217
Example system a	**3.82410**	5.21521
Example system ab	**2.78335**	4.90980

Table 2: The top fitness results of the driven PSO against the single PSO.

and how to appropriately choose settings, the latter has only two settings. Experimentation suggests that the PSO metaheuristic offers the best compromise between speed of convergence and satisfaction of requirements. The heuristic's optimisation method manages the candidate search, how the candidates communicate, how the candidates are created and mutated, but the fitness function defines the search. The capacity planning wizard enables and supports the user to define an appropriate fitness function.

Future work can proceed in a number of directions. For example, we aim to generalise the tool to work with other solvers within the PEPA Eclipse Plug-in tool suite. Whilst this is likely to be much more computationally expensive, it will be suitable for a broader range of models. Currently activity rates can only be searched through cloning subpopulations of components with different rates, but in the future we will extend the support for activity rates in the search parameter space. Furthermore, we will also allow more general specification of performance targets to be logic-based.

Acknowledgement

This work is partially supported by the EU project QUANTICOL, 600708.

References

1. Hillston, J., Tribastone, M., Gilmore, S.: Stochastic Process Algebras: From Individuals to Populations. Computer Journal **55**(7) (2012) 866 – 881
2. Marco, D., Cairns, D., Shankland, C.: Optimisation of process algebra models using evolutionary computation. In: IEEE Congress on Evolutionary Computation. (2011) 1296–1301
3. Marco, D., Scott, E., Cairns, D., Graham, A., Allen, J., Mahajan, S., Shankland, C.: Investigating co-infection dynamics through evolution of Bio-PEPA model parameters: A combined process algebra and evolutionary computing approach. In: Computational Methods in Systems Biology, LNCS 7605. (2012) 227–246
4. Karaman, S., Shima, T., Frazzoli, E.: A process algebra genetic algorithm. IEEE Transactions on Evolutionary Computation **16**(4) (2012) 489 – 503
5. Geisweiller, N.: Finding the Most Likely Values inside a PEPA Model According to Partially Observable Executions. PhD thesis, LAAS (2006)
6. Cerotti, D., Gribaudo, M., Piazzolla, P., Serazzi, G.: Asymptotic behaviour and performance constraints of replication policies. In: Proceedings of PASM 2014
7. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (2005)
8. Tribastone, M., Gilmore, S., Hillston, J.: Scalable differential analysis of process algebra models. IEEE Transactions on Software Engineering **38**(1) (2012) 205–219
9. Tribastone, M., Duguid, A., Gilmore, S.: The PEPA Eclipse Plugin. Performance Evaluation Review **36**(4) (2009) 28
10. Poli, R., Kennedy, J., Blackwell, T.: Particle swarm optimization; an overview. Swarm Intelligence (1) (2007) 33
11. Luke, S.: Essentials of metaheuristics, Lulu (2011)
12. Williams, C.: A capacity planning tool for PEPA. Master's thesis, School of Informatics, University of Edinburgh (2014)