Automatic Parameter Optimisation of Service Quality and Resource Usage [*]

Jacopo Mauro[1,2], S. Lizeth Tapia Tarifa[1] and Ingrid Chieh Yu[1]

[1] Dept. of Informatics, University of Oslo, Norway

{jacopom,sltarifa,ingridcy}@ifi.uio.no

[2] Dept. of Mathematics and Computer Science, University of Southern Denmark

Abstract

Developers use models to design real world distributed applications that often are subject to Service Level Agreements to find a good balance between the quality of the service and its resource usage. Executable models has been used to observe and study such applications using, e.g., the Real Time ABS language, an executable and object-oriented modelling language.

For complex models, due to the high number and dependencies between the parameters, it is very difficult to understand the best possible setting that leads the system towards a desired quality of service, while minimising the usage of computing resources. In this work we present POPT, a parameter optimiser tool that starting from Real Time ABS models, by using AI techniques, searches in an automatic way for the best possible setting to satisfy the developer's expectations.

1 Introduction

As part of the software development cycle, software engineering uses techniques such modelling and predictions to design and validate the applications to be developed, in such a way that both the clients and the development team have better certainty that the software to be implemented meets the desired expectations. A model of a software application is mainly a simplified representation of the real world application and, as such, only includes some aspects of interest that developers want to highlight and investigate. In this context, modelling is used to understand existing applications that need to be replaced, redesigned or improved, or to design new applications. When modelling software architectures, ideally the model includes design decision which take into account requirements related to quality of service and resource usage. For example, the application must have an acceptable level of response time and must adapt to a variable number of users with unexpected changes in their workload.

To predict that a software application will comply with some expected quality of service, we need to consider its deployment, together with other parameters that have an impact on its performance. For this, a model can be used to study how to put together these design decisions to reach the desired quality of service early in the software development process. This will allow to design adaptive applications, that can for example change according to its customer traffic while trying to keep a good quality of service with its current available resources. In particular, executable models have been used

[*] Supported by *SIRIUS - Centre for Scalable Data Access* (www.sirius-labs.no).

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

to describe and predict behaviours of software applications taking into account non-functional properties by using simulation techniques [1, 15, 20]. These case studies have used the formally defined Abstract Behavioural Specification language called Real Time ABS [12, 16]. This language realises a separation of concerns between the cost of execution and the capacity of available resources. A simulation tool for Real Time ABS supports rapid prototyping and visualisation. The use of modelling languages such as ABS thus enables developers to investigate parameters that usually impact quality of service decisions late in the software development at an early stage of software design.

After model creation, however, deciding good values for the parameters, is far from being a trivial activity. The parameters's values may have dependencies or have unexpected impacts on the behaviour of the overall application. Often, especially for complex models, the number of parameters is high, and each parameter may have a wide range of values to consider. A complete exploration of the entire configuration space is therefore unfeasible. In these cases, modellers usually use their expertise to select promising values for the parameters, but they may have bias, thinking that certain values are more promising than others and neglect to explore part of the configuration space that could in reality be more promising. Moreover, they may not have a full understanding of the trade-offs between the different parameters. Empirical evidence shows that using automatic configurators such as [10, 11, 17] can lead to better configurations.

In this work we describe an automatic parameter optimiser called `POPT`. The optimiser takes as input an executable model written in Real Time ABS to automatically explore configurations with the aim of finding promising values for the model parameters that will lead to satisfy the desired service quality at a minimum resource usage.

Paper Overview. Section 2 briefly introduces the Real Time ABS language. Section 3 contains the modelling of a running example to showcase the usage of `POPT`. Section 4 details the workflow of the tool, how to set it up and how to run it. Section 5 presents the results for the running example. Finally, Section 6 discusses related work and concludes the paper.

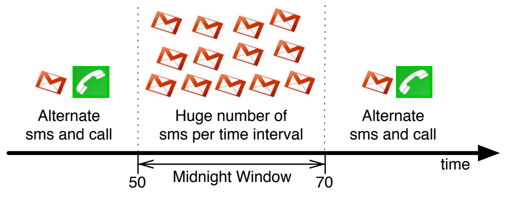
2 The Real Time ABS Language

Real Time ABS[1] is a modelling language for distributed and object-oriented systems with explicit deployment decisions. The main characteristics of ABS can be listed as follows: i) it has a formal syntax and semantics, ii) it cleanly integrates concurrency and object orientation based on concurrent object groups (COGs) [12, 24], iii) it supports both synchronous and asynchronous communication [5, 13], and iv) it offers a range of complementary modelling alternatives by integrating a functional layer with algebraic datatypes and functional programming, with an imperative layer [12] with COGs and asynchronous communication.

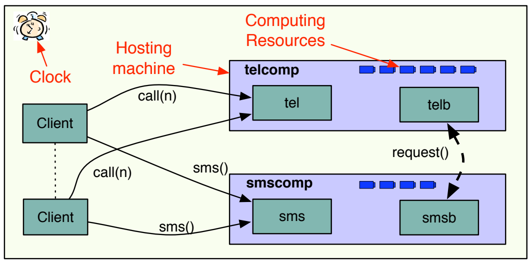
The functional layer of ABS is used to model computations on the internal data of objects. It allows designers to abstract from the implementation details of imperative data structures at an early stage in the software design. ABS includes a library with predefined datatypes such as Bool, Int, String, Rat, Unit, etc. It also has parametric datatypes such as lists, sets and maps. All other types and functions are user-defined.

The imperative layer of ABS allows designers to express interactions between concurrent object groups (COGs), which have an active thread of execution and communicate asynchronously via mailing boxes. Processes are encapsulated inside

[1] `http://abs-models.org/`



(a) The New Year's Eve client scenario.



(b) The telephone company scenario.

Figure 1: A scenario capturing two telephone services and their client handset.

COGs [12,24], and they are created automatically at method call reception and terminated after the method finishes its execution. ABS combines active (with a run method which is automatically activated when an object is created) and reactive behaviour of objects. ABS is based on cooperative scheduling: inside COGs, processes may suspend at explicitly defined scheduling points, at which point control may be transferred to another process. Only one process is active inside a COG, which means that race conditions are avoided.

Real Time ABS [3] extends ABS with modelling of the passage of dense time and deadlines to method calls. The time extension allows to represent execution time inside methods. The local passage of time is expressed in terms of a statement called **duration** (as in, e.g., UPPAAL [18]). Time values capture points in time during execution. Deadlines can be used to measure performance by checking weather a method was executed within an expected deadline. Deadlines are given as an annotation by the caller, when the method is called asynchronously.

Deployment is modelled using *deployment components* [16]. A deployment component is a modelling abstraction that captures locations offering (restricted) computing resources. The language also supports cost annotations associated to statements to model resource consumption. The combination of deployment components with computing resource and cost annotations allows modelling implicit passage of time. Here time advances when the available resources per time interval have been consumed.

ABS is supported by a range of analysis tools (see, e.g., [1]); in this paper we are using the simulation tool which generates Erlang code, as part of the automatic optimisation process to retrieve the quality of the simulation and its visualisation capabilities.

3 Example: mobile phone services on New Year's Eve

In this section we present a simple example to showcase the usage of POPT. The example, first introduced in [14] and further developed in this paper, is inspired by mobile phone users behaviour on New Year's Eve. In this time of the year, as depicted in Fig. 1a, people instead of alternating between making phone calls and sending SMS, flood the SMS server with messages during the so called "midnight window".

Let us assume that calls and messages are handled by two components, a Telephone and SMS server. The services are deployed on dedicated hosting machines and

Figure 2: The mobile phone services modelled in Real Time ABS

```
interface TelephoneServer {Unit call(Int calltime); ... }
class TelephoneServer implements TelephoneServer {
  Int callcount = 0; Int callsuccess = 0;
  Unit call(Int calltime){
    Bool result = durationValue(deadline()) > 0;
    while (calltime > 0) {
      [Cost: 5] calltime = calltime − 1;
       await duration(1, 1);}
    if (result) {callsuccess = callsuccess+1; ...}

interface SMSServer {Unit sendSMS(); ...}
class SMSServer implements SMSServer {
  Int smscount = 0; Int smssuccess = 0;
  Unit sendSMS() {
    [Cost: 1] smscount = smscount + 1;
    Bool result = durationValue(deadline()) > 0;
    if (result) ...}
```

interact with a number of clients, as depicted in Fig. 1b. The abstract implementations of the services in Real Time ABS are given in Fig. 2. The telephone server offers a method `call` which is invoked synchronously (i.e., caller waits for the callee), with the duration of the call as parameter. The SMS server offers a method `sendSMS` which is invoked asynchronously (waiting is not needed). Cost are added for each time interval during a call and for each sendSMS invocation to underline that handling calls and messages consume processor clock instructions. Calls and messages have a deadline, specified by the client, that should be met if possible. The deadline for a call is met if the call is started before the deadline, while for SMS the deadline is met if the message is sent before it.

```
class Handset (Int cyclelength, TelephoneServer ts, SMSServer smss) {
  Bool call = False;

  Unit normalBehavior() {
    if (timeValue(now()) > 50 && timeValue(now()) < 70) {
      this!midnightWindow();}
    else { if (call) { [Deadline: Duration(5)] await ts!call(1); }
          else { [Deadline: Duration(5)] smss!sendSMS(); }
    call = ¬ call; await duration(cyclelength,cyclelength);
    this!normalBehavior(); } }

  Unit midnightWindow() {
    if (timeValue(now()) >= 70) {this!normalBehavior();}
    else { Int i = 0;
      while (i < 10) {
         [Deadline: Duration(5)] smss!sendSMS(); i = i + 1;}
      await duration(1,1); this!midnightWindow(); }}

  Unit run(){this!normalBehavior(); } }
```

Figure 3: The client handset in Real Time ABS.

The model of the handset clients is given in Fig. 3. The handset makes requests to the two servers. The normal behaviour of the handset is to alternate between sending an SMS and making a call at each time interval. When it makes a call, the client waits for the call to end before proceeding. The handset's spike occurs between the time window starting at time 50 and ending at time 70 that represents the "midnight window". During the spike, the handset asynchronously sends 10 SMS requests at each time interval. To evaluate if it is the "midnight window", the expression `timeValue(now())` is used to check the current time.

```
interface Balancer { Unit request(DC comp);
                     Unit setPartner(Balancer p);}

class Balancer(String name) implements Balancer {
  Balancer partner = null;

 Unit setPartner(Balancer p) {partner = p;}

 Unit run() {
  await partner != null;
  while (True) {
   await duration(1, 1);
   InfRat total = await thisDC()!total(Speed);
   Rat ld = await thisDC()!load(Speed, 1);
    if (ld > overload()) { await partner!request(thisDC());}}}

 Unit request(DC comp) {
  InfRat total = await thisDC()!total(Speed);
  Rat ld = await thisDC()!load(Speed, 1);
  if (ld < underload() && finvalue(total)>20) {
   thisDC()!transfer(comp, finvalue(total)/mr(), Speed); }}}
```

Figure 4: The balancer in Real Time ABS

The model also consider resources and includes dynamic load balancing, which enables the two machines hosting the telephone and SMS servers to exchange resources. This is captured by the `Balancer` class in Fig. 4, whose instances run on each service. The `Balancer` class implements an abstract balancing strategy, transfers resources to its partner virtual machine when receiving a `request` message, monitors its own load, and requests assistance when needed. The behaviour of the load balancer depends on the parameters moving ratio `mr()`, `overload()`, and `underload()` which determine how much to transfer in each request and when to consider that the system is overloaded or underloaded, respectively.

The Balancer class takes as parameter a name (for friendly console-messages). The class has an active process defined by its run() method, which monitors the local load. The ABS default construct `thisDC()` returns a reference to the hosting machine on which an object is deployed and the method call `load(Speed, 1)` returns the usage (in percentage) of processing speed in the previous time interval. If the load is above `overload()` the balancer requests resources from its partner. If a `Balancer` receives a request for resources, it will consider the `underload()` parameter and transfer part of its available computing resources to its partner while maintaining its own capacity above a minimum.

```
{ // Main block:
  DC smscomp = new DeploymentComponent("smscomp", map[Pair(Speed, sms_rsc())]);
  [DC: smscomp] SMSServer sms = new SMSServer(); [DC: telcomp] TelephoneServer tel = new TelephoneServer();
  DC telcomp = new DeploymentComponent("telcomp", map[Pair(Speed, tel_rsc())]);
  if (with_balancers()==1){
    [DC: smscomp] Balancer smsb = new Balancer("smsb"); [DC: telcomp] Balancer telb = new Balancer("telb");
    await smsb!setPartner(telb); await telb!setPartner(smsb);}
  new Handset(1,tel,sms); ...// 30 clients
  println("succ,"+ toString(success)); ...// print the desired outputs to be used as part of the inputs for POPT
}
```

Figure 5: The main block configuration Real Time ABS

The configuration of the system is given in the main block of Fig. 5. Here we use deployment components to model the hosting machines and we decide which objects to deploy inside the machines using the `[DC: id]` annotations. The parameters `sms_rsc()`,`tel_rsc()` establish the amount of computing resources for each hosting machine per time interval. The parameter `with_balancers()` enables instead the usage of the balancers to check if their usage improve the quality of the system.

During the model execution the number of messages or calls that are delivered or started before the deadline are considered as successes, and, as shown in the next section, used to evaluate the quality of service provided by the system.

4 The Parameter Optimiser tool (POPT) for Real Time ABS

In this section we first describe the workflow of POPT, and later we describe the input required by POPT, how it works, and how it can be deployed. POPT is open source and freely available [2].

POPT's workflow. POPT uses as a back solver the Sequential Model-based Algorithm Configuration (SMAC) [10]. As depicted in Fig. 6, SMAC is used in a cyclic workflow performed by POPT where: i) the standard machine learning algorithm Random Forest (RF) is used to learn a RF model that relates the parameters of the ABS model with their quality, ii) the RF model is used to select new promising values of the parameters to try, iii) the new values are tested by running the ABS model, and iv) the output of the simulation is parsed to retrieve the quality of service. This information will then be used in the following iterations to refine the RF model, thus potentially having a more accurate estimation of the relation between the quality of the parameters and the parameters of the ABS model. This cycle can be interrupted after a given interval of time, after a given number of simulations have being performed, or when a given quality of solution has been reached. Once terminated, the best parameters which are used to configure the model are returned.

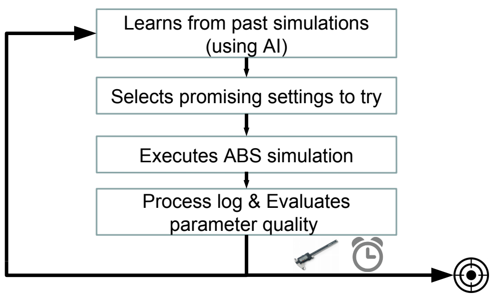

Learns from past simulations (using AI)
Selects promising settings to try
Executes ABS simulation
Process log & Evaluates parameter quality

Figure 6: POPT internal workflow.

[2] POPT can be downloaded from: https://github.com/HyVar/abs_optimizer/

POPT's input. As shown in Fig. 7, `POPT` requires three kind of inputs provided in various files: i) the ABS model, ii) the definition of the parameters, and iii) the desired target or metric that the modeller wants to reach/minimise.

Starting with the ABS model, the program should compile and run correctly in the Real Time ABS Erlang simulation tool, and it should expose the parameters that the modeller wants to tune with `POPT`. The parameters are expressed as constants in the ABS language with the syntax: **def** Int `p()` = v, where `p` and v are the name and value of the parameter, respectively.

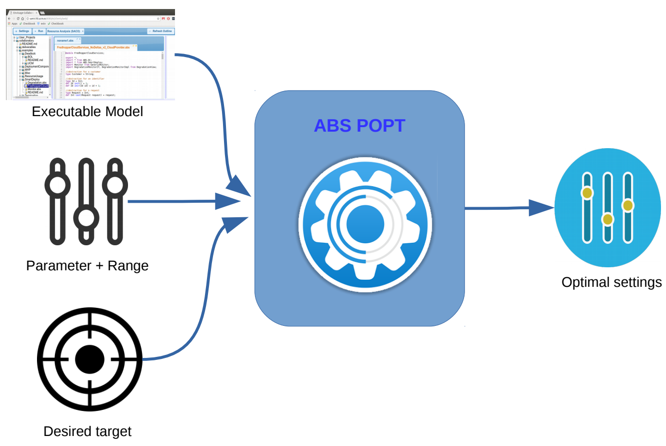


Figure 7: `POPT` inputs and outputs.

For the second input, `POPT` requires to associate to every parameter in the model, a finite domain of values that it can take. This is done following the standard syntax adopted by the SMAC optimiser. For the moment `POPT` only supports integers. Enumerations, such as a finite domain of string values, can be easily encoded by mapping their domain into a set of integer values. The parameters are defined in the file `params.pcs`.

Following our example, let us consider, e.g., the parameter `mr` with range [2,10] and a starting default value of 3. This can be defined with the following syntax:

```
mr integer [2,10] [3]
```

Similarly, for the parameter `with_balancers`, we want to explore the possible values `0` and `1`. This can be encoded as follows:

```
with_balancers integer [0,1] [1]
```

It is also possible to require the satisfaction of some constraints or to relate some parameters using Boolean operators, e.g., `{overload > underload}`.

The last ingredient for running the optimiser is to specify what is the metric to optimise. `POPT` tries to find out the best parameters maximising the, so called, *quality* of a simulation (i.e., a modelling abstraction of the quality of service). Therefore, it is vital to provide a function that associates the output of a simulation with its quality. This function is defined by means of a python program in the file `parse_abs_output.py`. In the python program, the user should define the code to parse the output of the simulation and return a real number that represent the quality of the simulation, where *the lower the value, the higher the quality*. The function that computes the quality is called `compute_quality` and takes as input the output generated by running the ABS model. SMAC will try to evaluate configurations where the number returned by the `compute_quality` function is smaller and smaller. In case of errors (e.g., when the ABS simulation ends prematurely or with an exception) the crash is captured and the quality of the simulation is assigned to a very large value (e.g., 10^9). In our example, let us assume that we want to tune the parameters to meet the following SLA: "*The application should have a success rate greater or equal than 85% for the response time of its services, and minimise the amount of computing resource usage*". Figure 8 presents a snippet of the compute_quality function to evaluate the quality of a simulation. We extract from the output the total number of invocations to both services and the number of successful invocations (i.e., invocations that meet their deadline) to define the success rate. If this value is greater or equal to 85% then we return the amount of resources used for both services, otherwise we return a large amount minus the success rate (indicating that the success rate is not reached). If the success rate is reached, POPT will in the next step try to minimise the resources, otherwise it will try to maximise the success rate.

Note that POPT supports also non-deterministic ABS programs, where the quality of the simulation may vary run by run due to the inherent concurrency of the model itself. In this case, POPT decides autonomously if the same configuration has to be simulated again, trying to maximise the average quality of the simulations performed for the same configuration.

```
... successrate = 100*float(success)/total
if successrate >= 85: return sms_rsc + tel_rsc
else: return 1100 − successrate
```

Figure 8: Computing the quality of the simulation of the example.

POPT's deployment and execution. To facilitate the deployment of POPT, we use the Docker container technology (see the github repository of POPT for the details on how to retrieve the image and run it). The container image comes with all the software and dependencies already installed, including python for parsing the output of the ABS program, Java to compile the ABS program and run SMAC, and Erlang to run the ABS simulations. To use it, the tool will need the previous specified files plus a file named settings.py to control the ABS simulations and a file named scenario.txt to control the entire tool execution. The file settings.py is used to establish constraints on the execution time of every simulation, specifying which are the ABS program files that define the model and how many times to repeat the simulation if it ends up in failure. The file scenario.txt is used to specify when POPT should terminate. In particular, two settings have to be considered: i) numberOfRunsLimit that limits the number of successful sequential simulations, and ii) wallClockLimit that limits the overall time POPT runs. With these two parameters POPT can be interrupted as soon as a given time is passed or a given amount of simulation are tried. When all these files are setup, the user can run the tool by invoking the command run_smac.sh.

POPT allows two forms or parallelisation: i) each ABS simulation can be speed up by running it on more than one core, and ii) multiple simulations can be run in parallel. In the latter case, following the recommended practice of SMAC, instead of executing a single instance of the configurator optimiser, POPT runs in parallel different execution of the configurator optimiser. The various executions can therefore proceed in parallel and share their results.

```
... Best solution found with quality 247.0
mr='2', overload='83', sms_rsc='142', tel_rsc='105',
underload='53', with_balancers='1'
Positive Runs 500, Crashes 0 ...
```

Figure 9: A possible output of the tool running the telephone company example.

When terminated, POPT logs all the configuration tried with their quality for every parallel run. To retrieve the best configuration tested and the total number of tests, the user can invoke the utility merge_states.sh. The output of the optimisation process is as shown in Fig. 9.

5 Example: optimising the resource usage of the services

We have applied POPT to the example in Sec. 3 and run the optimisation process in three different platforms: i) a Virtual Machine (VM) in a OpenStack based private cloud

Device	Modality	Time	Confs (unique)	Best Value
OpenStack VM	Non-Det	1h	453 (98)	251
OpenStack VM	Non-Det	24h	11084 (6886)	246
Numascale	Non-Det	1h	1111 (293)	248
Numascale	Non-Det	24h	25818 (9583)	248
Laptop	Non-Det	1h	592 (212)	247
OpenStack VM	Det	1h	489 (489)	252
OpenStack VM	Det	24h	10934 (10934)	243
Numascale	Det	1h	1105 (1105)	247
Numascale	Det	24h	18384 (18384)	242
Laptop	Det	1h	500 (500)	254

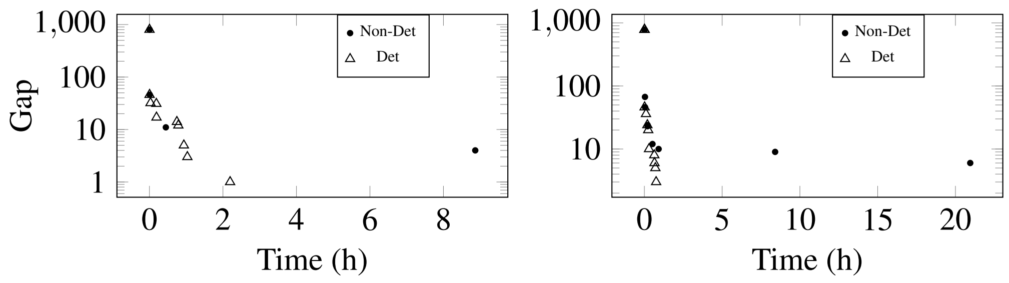
Figure 10: Summary of results of applying the tool in the running example.

running a Ubuntu 17.10 operating system, with 4 virtual cores and 8 GB RAM, ii) a Mac laptop with macOS Sierra v10.12.6. Processor 2,2 GHz Intel Core i7 and 16 GB RAM, and iii) a Numascale cluster [3] running CentOS 7.3 operating system (we used 8 nodes, each having 6 cores). We have run `POPT` for one hour in these three platforms. We have also run it one entire day on the OpenStack VM and on the Numascale cluster to study the improvement on the solution quality. For the laptop and the VM we used the `POPT` in sequential mode, and we used Docker to install it, as explained in Sec. 4. For the cluster we run 8 simulations in parallel, one for every node at disposal. Here `POPT` was installed from sources, since we did not have root privileges required to run Docker.

It is worth to mention that even though the original model is non-deterministic by nature, we initially decided to applied the `POPT` as if the model were deterministic. We did this because the deterministic execution modality allows to explore more unique configurations since it does not repeat the same run more than one time. During our experiments we realise that repeating the non-deterministic simulation may give different results of the quality function and therefore it may be possible that for simulations that almost reach the desired value (85 for our example), running the configuration only once is not enough. Indeed, when assuming that our example was deterministic, the best value obtained was `242` found by using the Numascale cluster after 5600 seconds. Note that in this solution, according to our quality function, 85 expresses the percentage of required success rate in the SLA, and 242 expresses the sum of the computing resources in both hosting machines (`sms_rsc+tel_rsc`), which according to the SLA we want to minimise. The parameters found for this solution are: `mr='2', overload='94', sms_rsc='142', tel_rsc='100', underload='54', with_balancers='1'.` Unfortunately, for this solution, the success rate is very close to `85`, and when the simulation is repeated more than once, it may happen that some of the runs give a success rate lower than `85` and therefore the quality function (cf., Fig. 8) returns a value greater than `1000`.

Taking into account the non-deterministic nature of the model and running more than one simulation for the same configuration during the optimisation process gives instead the following result: the best value of `246` was found after 31916 seconds by the VM OpenStack with the parameters `mr='2' overload='95', sms_rsc='154', tel_rsc='92', underload='46', with_balancers='1'.` `POPT` decided

[3]`https://www.numascale.com/`



(a) OpenStack VM. (b) Numascale.

Figure 11: Best solutions found over time. The y logarithmic axis represent the gap (either over 1000 or below 85, see Fig. 8) of the best value w.r.t. the global best value that the optimiser found (242).

to run this configuration 2000 times.

We believe that POPT can be extremely useful to try to balance the number of repetitions to have a reliable measure of the solution quality. If the exploration were done manually, a user could have found by luck the value of the parameters in a configuration to reach the value of 242, not noticing that this solution has a success rate so close to 85%, and realise later that due to the non-determinism of the model another execution would have violated the desired SLA.

We summarise the optimisation results in Fig. 10. For each run we reported the execution modality, the number of runs with and without unique configurations (this because non deterministic and parallel executions could run more than one simulation for the same configuration), and the value of the best solution that was found. As observed in the results, POPT is already effective when used with a normal laptop. Clearly, the more resources are available, the better the solution. When more CPUs are available, various simulations may run in parallel and, even if the CPUs are not very fast[4], a lot of configuration can be explored, in particular with the deterministic execution modality. However, in the case of a non-deterministic execution modality, the parallel execution may run a lot of repetitions for non-optimal solutions. This may hinder the best solution to be reached, but may in turn also increase its robustness.

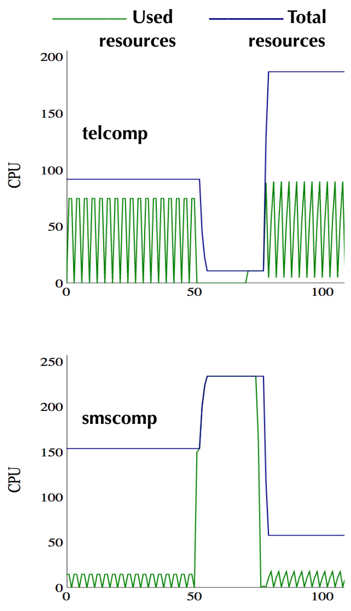


Figure 12: A simulation of the running example.

Figure 11 presents plots that describe how the value of the quality function has been reduced over time in the one day executions. In particular, we present the points in time when POPT has found a better solution both for the non-deterministic and deterministic modalities. The y axis shows the difference between the metric value at a given point in time w.r.t. 242 resources (sms_rsc+tel_rsc), which was the best value that POPT found that keeps the 85% of success rate.

As it can be observed, POPT can in a short amount of time reduce the returned value

[4]In the Numascale cluster, just the compilation of the ABS Program required more than 20 seconds, against less than 5 seconds on the VM.

of the quality function and find solutions which have a success rate of more than 85%. Finding further improvement by reducing the amount of computer resources is a slow process. The deterministic execution modality found the best values early (e.g. in less than 3 h). The non-deterministic modality instead required more time: more than 8 h in the OpenStack cloud, 20 for the Numascale cluster. This is no surprising due to the extra simulations needed by the non-deterministic modality.

The results are promising. Without the need for the user to specify any information, the outcomes have shown for instance that the balancers are indeed important and how to configure the balancer to best handle the extra workload during the midnight window. Figure 12 shows a simulation of the example where we can observe the resource usage and the transfer of resources using the parameters proposed by the optimisation tool in the non-deterministic modality. Note that the resource capacity values could be interpreted together with historic data of real applications, as done in [1, 15, 20]. The interpretation of this example is out of the scope of this paper, since our purpose is to show how to use the tool with a simple example. In particular we can observe the effectiveness of the load balancer during the midnight window.

6 Related Work and Conclusion

The idea of using models to study the best possible settings to apply to a concrete system is probably as old as science. With the advent of computers and software, models and simulators have also been proposed to control and optimise the usage of resources in software applications, especially after the advent of cloud computing which allows the renting of computing resources as needed.

Giving an overview of all the modelling languages and attempts to use models to optimise software or other systems is beyond the scopes of this paper. In this context, we would just like to remark that, while in general, the modeller often tries to tune manually the parameter of a model, there are plenty of approaches that try to automate this task. For example, just looking at the cloud setting, different approaches have been adopted to optimise the resource consumption based, e.g., on reinforcement learning [4,26], k-means clustering algorithm [25], autoregressive moving average [23], queuing network [27], learning automata [21], and Markov Decision Process [2]. These approaches, as also advocated in [9], can be used to improve a running system making it adaptable to changes. With `POPT` our focus was instead to decide which settings can meet a desired SLA during the design phase of an application. This allows to take into account deployment decision in the early phase of the development of a system. We are not aware of other methods which address optimisation of service quality at design phase.

To achieve this, we borrow techniques and best practices studied in the field of Programming by Optimisation [8] that tracks back to the mid 70's with the early works on portfolio theory and algorithm selection [22] that have only recently been readily available due to the powerful optimisation using machine-learning techniques and the computational environments required to execute them. In particular, we used SMAC [10], a parameter configurator that is often applied to the configuration of complex solvers for NP-hard problems. A preliminary version of `POPT` has been already successfully applied in an industrial settings to reduce the cost of a cloud based framework to dispatch car software updates [19]. Other related papers such [6] presents a model-driven approach to optimise the configuration, energy consumption, and operating cost of cloud and [7] where an stochastic model and a convex optimisation solver are used to satisfy the application's developer objectives while optimising the resource usage.

References

[1] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, S. Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, 2014.

[2] Bahar Asgari, Mostafa Ghobaei Arani, and Sam Jabbehdari. An efficient approach for resource auto-scaling in cloud environments. *International Journal of Electrical and Computer Engineering (IJECE)*, 6(5), 2016.

[3] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[4] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw., Pract. Exper.*, 41(1):23–50, 2011.

[5] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *Proc. 16th European Symp. on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.

[6] Brian Dougherty, Jules White, and Douglas C Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28(2), 2012.

[7] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a IaaS cloud. In *ICAC*. ACM, 2012.

[8] Holger H. Hoos. Programming by optimization. *Commun. ACM*, 55(2):70–80, 2012.

[9] Geir Horn. A vision for a stochastic reasoner for autonomic cloud deployment. In *NordiCloud*, volume 826 of *ACM International Conference Proceeding Series*, pages 46–53. ACM, 2013.

[10] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011.

[11] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res. (JAIR)*, 36:267–306, 2009.

[12] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. *Proc. 9th Int. Symp. on Formal Methods for Comp. and Objects (FMCO 2010)*, vol. 6957 of *LNCS*, 142–164. Springer, 2011.

[13] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

[14] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Dynamic resource reallocation between deployment components. *Proc. Int. Conf. on Formal Eng. Methods (ICFEM'10)*, vol. 6447 of *LNCS*, 646–661. Springer, 2010.

[15] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. *Proc. 14th Int. Conf. on Formal Eng. Methods (ICFEM'12)*, vol. 7635 of *LNCS*, 71–86. Springer, 2012.

[16] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *J. of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.

[17] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - Instance-Specific Algorithm Configuration. *ECAI*, vol. 215 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2010.

[18] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[19] Jia-Chun Lin, Jacopo Mauro, Thomas Brox Røst, and Ingrid Chieh Yu. A Model-Based Scalability Optimization Methodology for Cloud Applications. *7th IEEE Int. Symp. on Cloud and Service Computing, IEEE SC2*, 2017.

[20] Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. ABS-YARN: A formal framework for modeling Hadoop YARN clusters. *Proc. 19th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2016)*, vol. 9633 of *LNCS*, 49–65. Springer, 2016.

[21] Khosro Mogouie, Mostafa Ghobaei Arani, and Mahboubeh Shamsi. A novel approach for optimization auto-scaling in cloud computing environment. *International Journal of Modern Education and Computer Science*, 7(8), 2015.

[22] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[23] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. *CLOUD*. IEEE, 2011.

[24] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. *European Conference on Object-Oriented Programming (ECOOP 2010)*, vol. 6183 of *LNCS*, 275–299. Springer, 2010.

[25] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *ICAC*. ACM, 2010.

[26] Zhiguang Wang, Chul Gwon, Tim Oates, and Adam Iezzi. Automated Cloud Provisioning on AWS using Deep Reinforcement Learning. *CoRR*, 2017.

[27] Lydia Yataghene, Mourad Amziani, Malika Ioualalen, and Samir Tata. A queuing model for business processes elasticity evaluation. *IWAISE*. IEEE, 2014.