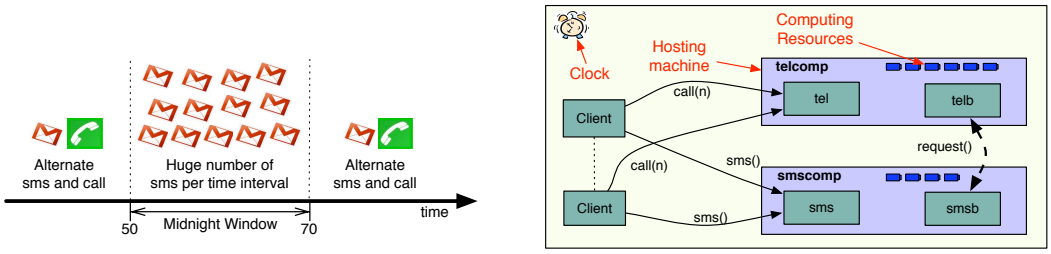

(a) The New Year's Eve client scenario.

(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.

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.

```
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 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.

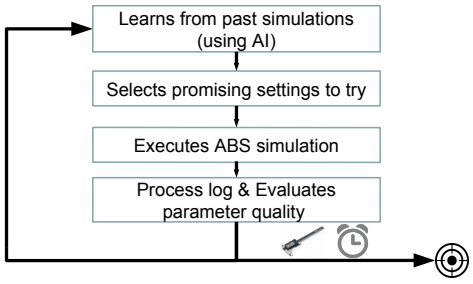

Figure 6: `POPT` internal workflow.

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