bcs

Release v1.0.1

Jun 22, 2022

Contents:

CHAPTER 1

Installation

bcs is hosted on Github. To clone bcs into your current directory, run

```
git clone https://github.com/MBoemo/bcs.git
git checkout v1.0
```

Navigate to the bcs directory and run

```
make
```

This will compile bcs and put a binary into the bcs/bin directory. bcs was written in C++11 and uses OpenMP for parallel computation. These are both standard on most systems and there are no other thirdparty dependencies. bcs was tested to compile on both Linux systems and OSX.

CHAPTER 2

Quickstart

To get something running and see how to use bcs, let's run one of the tests. Models should be written and saved in a separate file. These files are passed to the bcs binary on the command line. Navigate to the bcs directory and run:

```
bin/bcs -s 50 -t 1 -o firstSimulation tests/shouldPass/beacons-basic.bc
```

Here, we've specified the path to a model file (tests/shouldPass/beacons-basic.bc), the number of simulations we want to run (50), and the number of threads to use (1). There should now be a file called firstSimulation.simulation.bcs in your current directory. If we open this file, the first few lines should look like this:

```
>=======
0.015743        msg     proc2   j       3
0.151068        action2 proc2   j       3
0.160805        msg     proc1   j       3
1.14175e+06     longAction      proc1   j       3
>=======
0.202562        msg     proc2   j       3
0.231809        action2 proc2   j       3
0.315113        msg     proc1   j       3
303730  longAction      proc1   j       3
>=======
0.0960461       msg     proc2   j       3
0.11236 msg     proc1   j       3
0.229927        action2 proc2   j       3
569664  longAction      proc1   j       3
```

bcs output files all have the same format, where the start of each new simulation is marked with ">=======" and each row contains information about an action that a process performed during the simulation. From left to right, the tab-delimited columns specify the following information: the time when an action was done, the action name (if it was a non-messaging action) or channel name (if it was a handshake or beacon action), the process name that performed the action, the value of that process's parameters (if it has any) when the action was performed.

CHAPTER 3

Formatting

A bcs model, from top to bottom, consists of three sections:

- definition of static variables,
- definition of processes,
- system line (that is, the initial processes that are in the system at t=0).

In bcs, lines must be terminated by semicolons and any whitespace/newlines are ignored. If we want to define some variable x that will be used by a process, the three lines:

```
x = 5;
x                = 5;
x
=
5;
```

are all acceptable. Variables may be assigned either a float or an int, so assigning:

```
x = 3.452;
```

is acceptable. Variable names must either lead with an underscore or a letter. Thereafter, they can contain letters and numbers. However, leading with a number is not allowed. The following variable names are valid:

```
myVar
_myVar
__myVar
m1yVar99
```

But a variable name such as `99myVar` is not.

Comments are specified by `//` which tell the bcs parser that the rest of the line should be ignored.

CHAPTER 4

Processes

4.1 Process Definitions and Combinators

In the Beacon Calculus, components within a biological system are modelled as processes that can perform actions. An action is an ordered pair that specifies the action name followed by the rate. For example, we might define a process P as follows:

```
P[] = {exampleAction, 5};
```

This process can perform a single action called `exampleAction` at rate `5`, where rates are always the parameters of an exponential distribution. Once this process performs `exampleAction`, it cannot perform any other actions. It is therefore said to be deadlocked and is removed from the system.

A process that can only perform one action isn't particularly useful, especially for biological systems. We need a way to stitch series of actions together so that processes can perform complex behaviours. We define the following three combinators:

- Prefix `.`, where `P[] = {a,ra}.{b,rb}` is a process that performs action `a` at rate `ra` and then, once it has finished, performs action `b` at rate `rb`. Prefix is therefore used for actions that should happen in sequence.
- Choice `+`, where `P[] = {a,ra} + {b,rb}` is a process that makes an exclusive choice between performing action `a` at rate `ra` and action `b` at rate `rb`; it cannot perform both. Note that the probability of picking action a is equal to `ra/(ra+rb)`, so we can bias which outcome is more likely by scaling the actions' relative rates.
- Parallel `||`, where `P[] = {a,ra} || {b,rb}` is a process where actions `a` and `b` are performed in parallel at their respective rates.

Using these three combinators, we can now define more complex processes. In the following example,

```
P[] = {a,ra}.{b,rb} || {c,rc}.{d,rd} + {e,re}.({f,rf} + g,rg});
```

we make an exclusive choice between actions `c` and `e`. If we pick `c`, then we go on to perform action `d`. If we pick `e`, then we make another choice between `f` and `g`. All the while, in parallel, we perform action `a` followed by action `b`.

4.2 Parameters and Gates

Oftentimes, processes need to keep track of certain quantities. For example, if a process models the amount of a certain chemical reactant in a system, the process must be able to keep a count of how many molecules of this reactant are present over time. If a process models a DNA replication fork, it has to keep track of where the replication fork is on the chromosome. This is achieved through parameters, which are values that a process keeps track of. Parameters are specified between square brackets, and processes can increase or decrease the value of their parameters over time. They can also use the value of their parameters in the computation of rates.

Suppose there is a car which is at a particular location. We can express this as `Car[i]`, where the `Car` process has parameter `i` which specifies its location. We can specify movement of the car through recursion. The following process models a car that drives at rate 0.1, and increases its parameter value as it moves.

```
Car[i] = {drive,0.1}.Car[i+1];
```

This car, as it is modelled above, will keep driving without stopping. We may wish to specify, for example, that the car should stop when it reaches a bus stop at `i=10`. To express this, we use a gate:

```
Car[i] = [i < 10] -> {drive,0.1}.Car[i+1];
```

The gate in front of the drive action specifies that this action can only be performed if the gate's condition is satisfied. In this case, the value of the car's parameter `i` must be less than 10. If the car starts at `i=0`, then the car continues driving until `i=10` at which time the gate's condition is no longer satisfied. The car can no longer perform any actions, so the process deadlocks and the simulation stops.

In addition to the less than comparison used here, bcs supports the following logical operators:

- <=, less than or equal to,
- ==, equal to,
- `!=`, is not equal to,
- >, greater than,
- >=, greater than or equal to,
- `&`, logical and,
- `|`, logical or,
- `~`, logical not.

CHAPTER 5

The System Line

The initial state of a Beacon Calculus system is specified using a system line. Only one system line is allowed in the model file, and it must be the last line in any bcs model. The system line specifies the processes present in the system at time zero; specifically, how many of each process we have and their parameter values. For example, a complete piece of bcs source code is:

```
driveRate = 0.1; //static variable definition

Car[i] = [i < 10] -> {drive,0.1}.Car[i+1]; //process definition

Car[0]; //system line
```

The first line specifies a variable definition. The second line specifies the process definition for a car as we've done previously. The third and final line is the system line which specifies that at time zero, there is one copy of a car process in the system and the value of its parameter is `i=0`. Suppose instead we wanted 50 cars, each of which start at `i=0`. Then we can write:

```
driveRate = 0.1;
initialCars = 50;

Car[i] = [i < 10] -> {drive,0.1}.Car[i+1];

initialCars*Car[0];
```

If we wanted 50 cars to start at `i=0` and 20 cars to start at `i=5`, we can write:

```
driveRate = 0.1;
initialCars = 50;

Car[i] = [i < 10] -> {drive,0.1}.Car[i+1];

initialCars*Car[0] || 20*Car[5];
```

CHAPTER 6

Communication

6.1 Handshakes

Processes need to be able to interact with one another. In doing so, they can change their actions in response to other processes. In the Beacon Calculus, two processes can communicate synchronously (at the same time) via handshake actions:

- A handshake send action `{@chan![i], rs}` sends value `i` on channel chan at rate `rs`.
- A handshake receive action `{@chan?[S](x), rr}` receives one of a set of values `S` on channel chan at rate `rr` and binds the result to `x`.

If the handshake happens, the handshake receive and handshake send actions happen together at rate `rs*rr`.

Let's illustrate with an example. A surveyor has been tasked to count the number of cars that travel down a road over time. They surveyor is located at `i=5`, and performs a handshake with each car as it goes by.

```
driveRate = 0.1;
initialCars = 50;
fast = 10000;

Car[i] = [i < 10 & i!= 5] -> {drive,driveRate}.Car[i+1]
       + [i==5] -> {@count![0],fast}.{drive,driveRate}.Car[i+1];
Surveyor[c] = {@count?[0],1}.Surveyor[c+1];

initialCars*Car[0] || Surveyor[0];
```

This model beings with 50 cars at `i=0` and a surveyor that has counted 0 cars. If a car isn't at `i=5` then it steps as normal. If the car is at `i=5` then it handshakes with the surveyor at a fast rate using channel count. Both the actions `{@count![0],fast}` and `{@count![0],fast}` happen simultaneously at rate `fast*1 = fast`. Once both a car and the surveyor perform the handshake, the surveyor increases their count by one by incrementing parameter `c` and recursing. The car carries on driving as normal.

In the above example, the handshake could receive a single value (5) on channel count. However, as mentioned above, handshakes can accept a set of values on a particular channel. Suppose the surveyor wants to keep a separate count of cars that are red. Consider the following model:

```
driveRate = 0.1;
nonRed= 25;
red= 25;
fast = 10000;

Car[i,r] = [i < 10 & i!= 5] -> {drive,driveRate}.Car[i+1]
         + [i==5] -> {@count![r],fast}.{drive,driveRate}.Car[i+1];
Surveyor[c,cr] = {@count?[0..1](x),1}.Surveyor[c+1,cr+x];

nonRed*Car[0,0] || red*C[0,1] || Surveyor[0,0];
```

The Surveyor process has two parameters, `c` and `cr`, which keeps track of the total count and red cars, respectively. Car has an additional parameter `r` which is either 1 (red) or 0 (not red). The range operator `..` in `{@count?[0..1](x),1}` means the set of all consecutive integers between 0 and 1 (inclusive). In this case, this is the set {0,1}. So the handshake receive action accepts one of two possible values and binds what it receives to `x` for later use. If the car it handshakes with is red, it receives value 1. When the surveyor process recurses, `x=1` so it increments the red car count `cr` by one. If the car was not red, then the value received would have been 0 so that `x=0` when the the process recurses. Therefore, `cr+x = cr+0 = cr` so the red car count is not increased.

In addition the range (..) operator used above, bcs supports the following set operations for both beacons and handshakes:

- `U`, set union,
- `I`, set intersection,
- `\`, set subtraction.

For example, the following Beacon Calculus operations (left hand side) correspond to these sets (right hand side):

```
-5..2 = {-5,-4,-3,-2,-1,0,1,2}
1U8..10 = {1,8,9,10}
1I8..10 = {}
1\8..10 = {1}
15..18\16 = {15,17,18}
0..2U8..15I4..9 = {0,1,2,8,9}
```

6.2 Beacons

Handshakes provide the means for synchronous communication between processes, whereby two processes each perform handshake actions at the same time. The Beacon Calculus also allows processes to communicate via beacons, which is asynchronous communication. Any process can launch a beacon that transmits a value on a channel. The beacon stays active until it is explicitly killed by a process (not necessarily the same process that launched it).

- A beacon launch, `{chan![i],rs}` launches a beacon that transmits value `i` on channel chan at rate `rs`.
- A beacon kill, `{chan#[i],rs}` kills a beacon (if there is one) transmitting value `i` on channel chan at rate `rs`.

Once a beacon is launched, processes can interact with active beacons in two ways.

- A beacon receive `{chan?[S](x),rr}` can only be performed if there is an active beacon on channel chan transmitting a value in set `S`. If there is such a beacon, a process can perform the beacon receive action and bind the value it receives to `x` for later use.
- A beacon check `{~chan?[S],rr}` is the inverse of a beacon receive. This action can only be performed if there is no active beacon on chan transmitting any value in `S`.

While a beacon is active, it can be received any number of times by any number of processes. Once a beacon has been killed, it can no longer be received.

Let's consider a simple example. Suppose there is a traffic light at `i=5` that switches between red and green. Cars can only pass through the intersection at `i=5` when the light is green. Otherwise, they have to wait.

```
driveRate = 0.1;
change = 0.001;
initialCars = 50;
fast = 10000;

Car[i] = [i < 10 & i!= 5] -> {drive,driveRate}.Car[i+1]
       + [i==5] -> {~red?[0],driveRate}.Car[i+1];
TrafficLight[g] = [g==1] -> {green#[0],change}.{red![0],fast}.TrafficLight[0]
                + [g==0] -> {red#[0],change}.{green![0],fast}.TrafficLight[1];

initialCars*Car[0] || TrafficLight[0];
```

In the above model, there is a process `TrafficLight` with a parameter `g`. When `g=1`, the traffic light is showing green. When `g=0`, the traffic light is showing red. If the traffic light is showing green, it keeps a beacon active on channel `green`. When the traffic light switches, it kills the beacon on `green` and launches a new one on channel `red`. Switching from red back to green is similar. In order for a car to move through the intersection at `i=5`, it performs a beacon check to make sure the light is not red. If the light is red, the car has to wait until the light turns green as it cannot perform the beacon check action while there is an active beacon on channel `red`.

In this example, we could have created a model where the traffic light handshakes with each car rather communicate via beacons. However, this would have been slightly more cumbersome. Beacons make it easy and concise to communicate a state change to a large number of other processes.

CHAPTER 7

Built-in Functions

While some of these functions and operators were mentioned and used above, this section lists those that bcs currently supports.

In rate expressions and gates:

- `max(x,y)`, the maximum of a and b,
- `min(x,y)`, the minimum of a and b,
- `abs(x)`, the absolute value of x,
- `sqrt(x)`, the square root of x.

In handshake receives and beacon receives:

- `..`, range,
- `U`, set union,
- `I`, set intersection,
- `\`, set subtraction.

In rate expressions, gates, and handshake/beacon receives:

- `+`, sum,
- `-`, difference,
- `*`, product,
- `/`, quotient,
- `^`, exponents.

CHAPTER 8

Simulation

8.1 Options

The bcs executable accepts a number of arguments to control the simulation of a Beacon Calculus model:

- `-t`, the number of threads. Simulations can be run independently on separate threads, so multithreading can speed up runtimes considerably. We recommend using as many threads as you have available if the simulation is large.
- `-m`, the maximum number of actions allowed before the simulation is stopped. If `-m 100` is specified, the simulation will stop (even if it is not deadlocked) after a total of 100 actions have been performed by processes in the system. In practice, this is useful for checking a model's behaviour.
- `-d`, time at which the simulation stops. If `-d 60` is specified, the simulation will end when the time is equal to 60, or before if the system has deadlocked.

8.2 Algorithm

Models are simulated using a modified version of the Gillespie algorithm, and are therefore subject to some of the algorithm's disadvantages. In particular, systems with long simulation durations and lots of high-rate actions can head to slow bcs runtimes. Ways to improve this are currently in development.

8.3 Casting

As a simulation runs, arithmetic functions can be applied to parameters, channel names, and sent/received values. Each of these can be either ints or floats. When a function acts on both ints and floats, the result is upcast to a float. For example in the following bcs code, process `P` starts with `i=0` (an int) but after it is multiplied by 2.0 in the first recursion, the type of `i` is a float thereafter.

```
//process definition
P[i] = {multiplyParameter,1.0}.P[i*2.0];

//system line
P[0];
```

In any operation where comparisons must be made (gates, handshakes, and beacons) values must be ints, and bcs will throw an error if passed a float. It is perfectly acceptable to use a float as a parameter, where it may be used to scale an action rate, etc.

8.4 Output

Simulation outputs are always written in a file with the `.simulation.bcs` extension and the prefix specified by the user using the `-o` flag. Each simulation begins with the line `>=======`. Each line is an action that was performed by a process in the system, and the tab-delimited columns, from left to right, specify:

- the time in the simulation when the action happened,
- the name of the action that was performed (or the channel name, if it was a beacon or handshake action),
- the name of the process that performed the action,
- the parameter values (if any) of that process at the time when the action was performed.

For example, the output line

```
0.315113   act1    P       i       2       j       5
```

indicates that an action named `act1` was performed by process `P` at time 0.315113. Process `P` has two parameters, `i` and `j`, and when this action was performed, `i=2` and `j=5`.

See *Quickstart* for a further example of the output file format.

CHAPTER 9

Plotting

During the process of developing bcs and applying it to biological problems, we found that in practice, the type of plots required for a given application were often nonstandard and more complicated than number-of-processes-over-time plots. Indeed, all of the examples in Beacon Calculus manuscript required bespoke plots that were typical of the applications' respective fields. For this reason, we focused mainly on making an output that was easy to parse (see *Simulation*) so that users could reshape the results into whatever plot was appropriate.

For completeness, testing, and simper applications, we included a plotting script bcs/utils/plot_bcs.py which uses matplotlib to plot the value of a process's parameter over time. Suppose we have the following model:

```
r = 1.0;
fast = 1000;

A[count] = {@react![0],r*count};
B[count] = {@react?[0],count}.{@gain![0],fast};
C[count] = {@gain?[0],fast}.C[count+1];

A[10] || B[10] || C[0];
```

Here, processes `A`, `B`, and `C` represent populations of distinct chemical species, where the molecules of each species is given by parameter `count`. `A` and `B` both start with populations of 10, and `A` and `B` can react at rate `r` to add to the population of `C`. We can save the above model as simple.bc can run five simulations of the model using bcs:

```
bcs -s 5 -t 1 -o abcsim simple.bc
```

which creates a file abcsim.simulation.bcs. We can plot the value of `i` in `C` over time:

```
python plot_bcs.py -a gain -p C -i count -o myplot.png
```

The resulting plot file `myplot.png` will have five traces, one for each of the five simulations run, showing the value over time of parameter `count` for process `C`.

CHAPTER 10

Examples

New users are encouraged to look in the bcs/tests directory, which contains dozens of very simple models with the expected output and behaviour described in the comments.

We included examples from different fields of biology in the main Beacon Calculus manuscript, which are:

- simple reaction between two chemical species (see section “Language Overview”),
- DNA replication in *S. cerevisiae* (see section “DNA Replication”),
- a population of cells responding to methylation damage (see section “Cellular Response to DNA Damage”),
- receptor multisite phosphorylation and ultrasensitivity (see section “Multisite Phosphorylation”),
- kinesin stepping down a microtubule (see Supplemental Information).

Full source code is provided for each of these examples, along with a line-by-line description of the processes and expected behaviour of the model. For convenience, the source code for these models are provided in the bcs/examples directory, along with python scripts that create plots similar to those in the manuscript.

As we develop more applications and examples over time, we will post them here.

CHAPTER 11

Overview

The Beacon Calculus is a process algebra designed to make modelling biological systems fast, intuitive, and accessible. It was designed with two main aims in mind: It should be possible to represent complex systems with minimal amounts of code, and models should be very easy to expand and modify. Once models are written in the Beacon Calculus, they can be simulated with the Beacon Calculus Simulator (bcs) software detailed here . Users interested in an introduction to the Beacon Calculus should first look through the corresponding methods publication which introduces the language by way of examples. A more formal description of the language and semantics is presented in the supplemental information of that publication. This page aims to catalogue the features of both the language and the software for reference.

CHAPTER 12

Publications

Please cite the following publication if you use the Beacon Calculus for your research:

Boemo, M.A., Cardelli, L., Nieduszynski, C.A. (2020) The Beacon Calculus: A formal method for the flexible and concise modelling of biological systems. *PLoS Computational Biology* 16:e1007651. [Journal DOI] [bioRxiv]

CHAPTER 13

Bugs, Questions, and Comments

Should any bugs arise or if you have any questions about usage, please raise a GitHub issue. For more detailed discussions about collaborations or the Beacon Calculus, the bcs software, or the documentation, please Email Michael Boemo at mb915@cam.ac.uk.