SPC builder practical manual

Francesco Rinaldi David Gilbert

Brunel University London

November 27, 2023

Abstract

The SPC code, utilised for implementing the Hybrid Approach-based ('Movement and communication in multiscale/multilevel models of biological systems", F. Rinaldi, D. Gilbert, M. Heiner and L. Ghanbar) models as mentioned in the main paper, facilitates entity interaction with Spike. However, it lacks support for certain traditional coding operators crucial for preventing redundant code sections. This issue is insignificant when modelling small spaces with few Complex Entities, but becomes more apparent when representing biological systems that typically require multiple Complex Objects interacting within a larger shared space by exchanging stimuli through the environment. The larger the space and the higher the number of Complex Entities involved, the more repeated SPC code lines are generally needed to verify each Complex Object's position within the environment, and to enable movement and interactions. The SPC Builder serves as a valuable extension of the Spike suites, implemented in Python, to streamline the process of generating repeated blocks of SPC code. Its core concept is simple: represent traditional SPC code with Python strings and utilise built-in coding operators and statements, such as loops, if statements, and arrays, to streamline the generation of SPC code. The SPC Builder, available at SPC BUILDER, significantly enhances the Spike suites by expediting the generation of repeated SPC code blocks, eliminating the need to write several and repeated lines of SPC code manually.

The SPC builder system was constructed as part of the activities of the research project "Movement and communication in multiscale/-multilevel models of biological systems" funded by the Leverhulme Trust under their Emeritus Fellowship Scheme (Project number EM-202-0-086\9).

Contents

List of Figures

1 Introduction

The SPC Builder is a tool designed to streamline the implementation of a Hybrid Approach for Petri net models that require the movement and interaction of Complex Objects. It automates the process of writing SPC code, making it easier to implement such models' mechanisms. The Leverhulme project technical report ("Movement and communication in multiscale/multilevel models of biological systems", F. Rinaldi, D. Gilbert, M. Heiner and L. Ghanbar) offers a theoretical perspective on the Hybrid Approach, while this manual aims to provide three practical examples that demonstrate how to effectively use the SPC Builder to automate the SPC code writing process for Hybrid Approach-based models. Gaining hands-on experience with the SPC Builder is crucial for successfully modelling complex systems using the Hybrid Approach. The examples presented in this manual encompass various structures, operands, and procedures essential for implementing the Hybrid Approach in your models. These examples serve as templates, illustrating the structure of the SPC Builder, its usage, the set-up for the Petri net models, and the appropriate way to write the SPC code for automated script generation. The three scenarios that have been adapted from the Leverhulme report are:

1. Whale Sample
2. Dicty
3. LioBiofilm

To fully comprehend this paper, familiarity with the Spike system and its associated SPC code is necessary. The SPC code used within the SPC Builder for each showcased scenario can be downloaded as a zip file from `https://github.com/rin0o1/SPC-python-builder`.

2 System Requirements

SPC builder is written in Python, and can be used on all platforms which can run Python. The code produced is used as commands to the Spike simulator; this runs on Windows (version 10 or greater), Mac OS X (10.12+, Intel only, 64bit) and Linux (Intel only, 64bit).

Utilising the SPC Builder necessitates a foundational knowledge of Petri net models (CANDL, ANDL), an understanding of SPC code and Spike, familiarity with the Hybrid Approach and its functionality, as well as basic Python coding skills. This paper will not dive into these topics; instead, it

will concentrate on the implementation and usage of the SPC Builder. To implement Hybrid Approach-based models, the installation of the following tools is required:

1. Snoopy (v2.5)
2. Spike (v1.6.0rc4) downloaded and added to the environment variables as *spike*
3. Python 3
4. The SPC-python-builder GitHub repository containing the code base for the builder cloned in your local machine

3 Software Structure

All the SPC builder projects consist of a set of Python classes:

- main.py: This serves as the entry point to initiate the builder. You will execute this file along with a set of parameters to run *BuilderANDL.py* and *BuilderSPC.py*, which are the two core Python classes responsible for automatically constructing the ANDL and SPC files.
- BuilderANDL.py: This file contains functions that transform a user-inputted CANDL file into an ANDL version for simulation. Within this class, the *editCandl* method reads the provided CANDL file and replaces specific comments with values passed as parameters when invoking the SPC Builder script. This mechanism enables the parameterisation of the model's colour set values. For example, in the scenarios presented in this manual, the "//dimensions" comment is replaced with the variables "DX" and "DY", and "//instances" is replaced with the variable "instances". The values for these three variables are determined by the user when setting the "–x", "–y", and "–o" parameters while executing the script. This allows for easy modification of the grid environment's dimensions and the number of Complex Objects or colour sets, which are fundamental components of the Hybrid Approach. The "dimensions" and "instances" comments can be changed by setting the *CANDL_COMMENT_DIMENSIONS* and *CANDL_COMMENT_NUMBER_OF_INSTANCE_COMPLEX_OBJECT* variables in the Constants.py file.

- BuilderSPC.py: This file is where the SPC script should be embedded as Python code. The Python code will be automatically converted into SPC code and exported to the *spcOut.spc* file for simulations. *BuilderSPC.py* comprises three primary methods: *onStepVariableDeclaration*, *onStepStepwise*, and *export*, which correspond to the three main sections of an SPC script.

    1. *onStepVariableDeclaration*: This method should contain the SPC code, encoded as Python strings, necessary for declaring all variables, constants, and observers required by the *onStep* object
    2. *onStepStepwise*: This method should consist of the SPC code, encoded as Python strings, that defines the block of rules to be executed for each timestep of the simulation within the *onStep* object.
    3. *export*: This method should include the SPC code, encoded as Python strings, that specifies the values of model markings, observers, and constants to be exported at the end of the simulation.

    Within each of these three functions, there is a string variable called spc. This variable will contain the SPC code encoded as a Python string, which will be written into the *spcOut.spc* file.

- BuilderHandler.py: executes BuilderANDL, and BuilderSPC

- BuilderOptions.py: A Python object stores all the parameters that users provide as inputs. Both *BuilderSPC* and *BuilderANDL* can access this object using "self.o".

- Constants.py: a Python object storing all the constants used within *BuilderANDL* and BuilderSPC

4 Using the SPCBuilder.py

Utilising this tool involves a series of common steps necessary for the implementation of the Hybrid Approach, as outlined in this section: setting up the Petri net model, configuring the CANDL file, and writing the SPC code. The SPC Builder assists with the final step, while the first two steps are essential for this new approach and must be completed regardless of whether the SPC Builder is used. The default code provided for the builder has been optimised for efficient implementation of the Hybrid Approach. However, this script can also be employed for any SPC scripts that contain repeated lines of

code, which can be condensed into fewer lines using standard programming language operators.

4.1 Setting up the Petri net model

The Hybrid Approach is distinguished by its ability to move Complex Objects within a grid environment. Models seeking to implement this approach using the SPC Builder require a Color Set for defining the grid dimensions and another for the Complex Object. Hybrid-based models are simulated through an SPC script that takes an ANDL file as input. The SPC Builder can generate these two files (from a CANDL file) in a semi-automated fashion to streamline the implementation process. To compute the file generation, the builder needs a CANDL file and a properly configured Petri net model, as demonstrated below. The correct model setup is essential for exporting the appropriate CANDL (Section 4.2), which the SPC Builder will use as input to generate the ANDL and SPC script semi-automatically. The colour definition specifying the number of instances for the Complex Object must be a Simple Color Set of type *int* (Figure 1). Upon unfolding, this value informs Snoopy of the number of instances to create for the model representing the Complex Object and its internal structure. All Colored places within the model shaping the Complex Object must have their Markings Colorset set with the respective Simple Color name (Figure 2).

The colour definition for the grid environment must be a Compound Color of type *product*, with its value determined by the product of two (2D space) or three (3D) Simple Colors that define the dimensions of the space (Figure 3). These Simple Colors, which define all available *x,y* or *x,y,z* coordinates in the grid space, must be of *int* type and set as min-max, indicating the value range for each axis. For instance, the Simple Colors used for the product of the Compound Color (Figure 4) defining a 9x9 grid space would be specified as reported in (Figure 5) within the ColorSet definition in Snoopy.

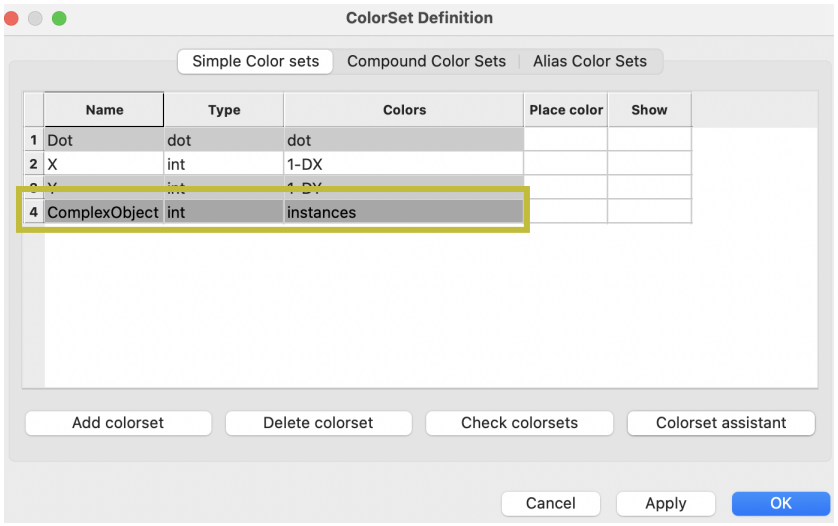
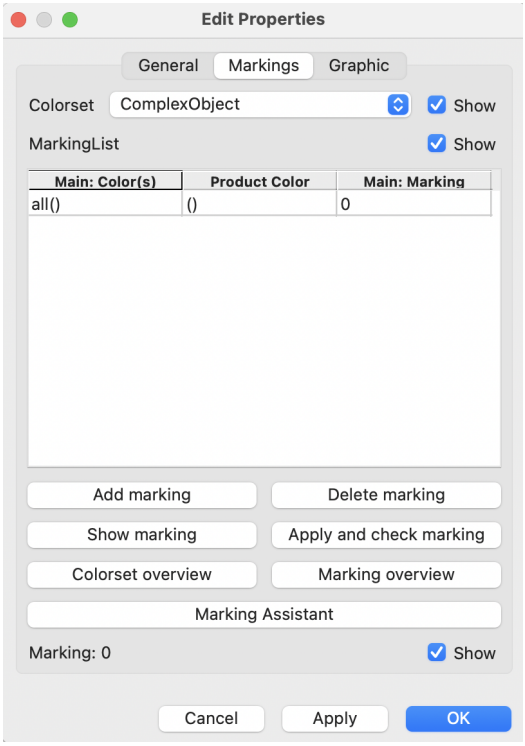


Figure 1: Snoopy Screenshot - *ComplexObject* is a Simple Color set of type *int* used for indicating the number of instances of the complex object to create on unfolding the model. Its value is dictated by the *instances* value which is a model constant.

Employing constants to define the Simple Color Sets streamlines the process of adjusting the number of instances and grid dimensions (Figure 6). By avoiding hard-coded values, changes can be made without the need to modify the entire model. For example, altering the constant *instances* will affect the number of *ComplexObject* colours, while adjusting the constants *DX* and *DY* will alter the size or shape of the grid environment. Moreover, using constants makes them easily accessible from the CANDL file. As the SPC Builder relies on the CANDL file, it can modify the number of Complex Object instances and grid dimensions by simply changing these constants.



S

Figure 2: Snoopy Screenshot - The Colorset marking for each Petri net place included within the Complex Object structure has to be set to *ComplexObject* as well as the Simple Colour set shown in Figure 1.

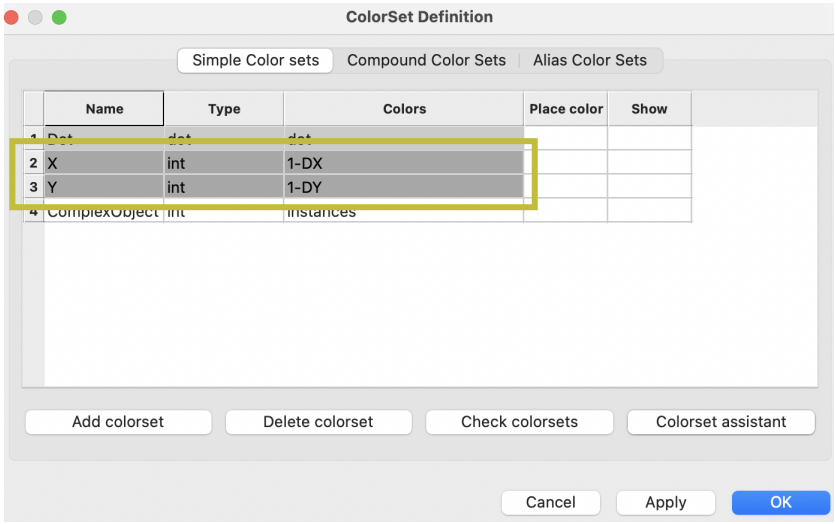


Figure 3: Snoopy Screenshot - X and Y are the *int* Simple Colour sets dictate the dimension of the Compound Color *Grid2D* (Figure 4). With the current set up X axe ranges from 1 to DX which is a model constant set to 9, whereas Y axe ranges from 1 to DY which is a model Constant set to 9.

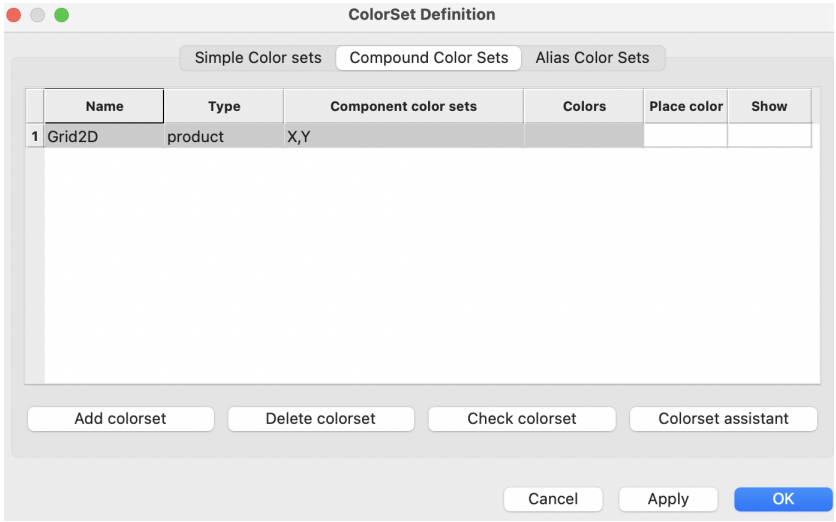


Figure 4: Snoopy Screenshot - *Grid2D* is a Compound Color of type *product* which values are defined by the product of the X and Y Simple Colour sets illustrated in Figure 3. *Grid2D* is the Colour name used for setting the Colorset within the Markings section of the Coloured Petri net model place that is used to encode the locations (Figure 5).

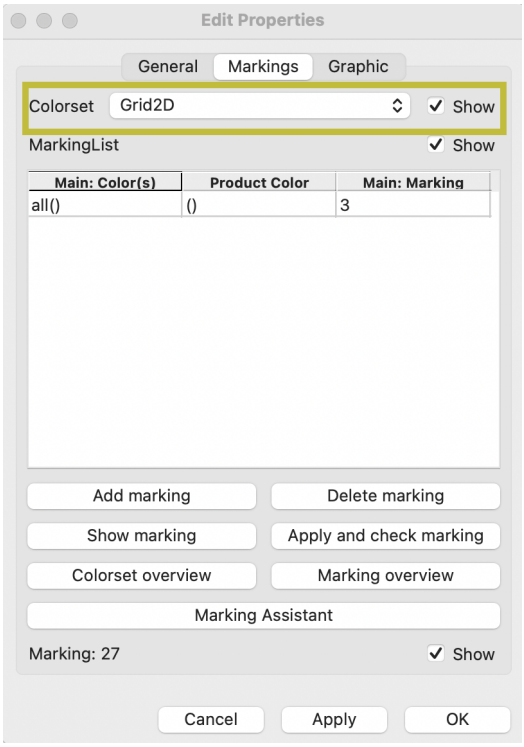


Figure 5: Snoopy Screenshot - Markings definition for the coloured place that is used to encode the grid environment

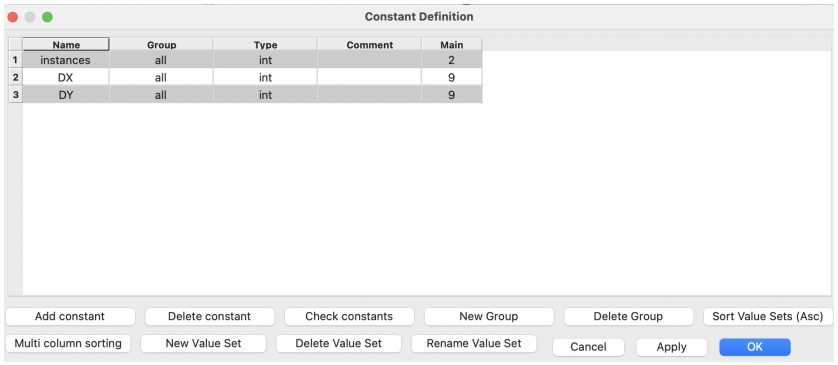


Figure 6: Snoopy Screenshot - *instances*, DX, and DY are the model constants used for defining the *ComplexObject*, X, and Y Color sets. On exporting the model as CANDL these constants will be available on the *parameter* section of the CANDL file itself.

4.2 Configuring CANDL

This section demonstrates some manual adjustments needed for the CANDL file obtained upon exporting the coloured model using Snoopy. The SPC Builder operates on CANDL files, but to generate the final SPC script with the SPC code for integrating the Hybrid Approach mechanism, it requires access to the unfolded Petri net model (ANDL file). To achieve this, the SPC Builder incorporates an internal routine that employs a system call to Spike to convert the CANDL file into an ANDL file, which is then used as input for the SPC script's simulation and *onStep* operations. One of the parameters refer to the parameters section for executing the SPC Builder is the path location of the CANDL file. Since CANDL files resemble machine language more than human language, it is feasible to construct external scripts that can interpret and modify them, as well as edit the model programmatically. The SPC Builder serves as an example of such an external script, allowing for the management of CANDL and SPC files using a standard programming language like Python. The constants declaration set in Snoopy (see Figure 6) is easily accessible from the *constants* section of a CANDL file. The SPC Builder leverages this section of the CANDL file to replace the values of the model's constants (*instances*, *DX*, and DY) with the values supplied by the user as parameters when running the script. This mechanism offers users flexibility and abstraction when working with the model, as they can modify the number of complex objects and grid dimensions by simply providing different parameters when invoking the builder, without needing to alter the model itself. By using this method, the unfolded model will represent the number of complex object instances and grid dimensions specified by the user when executing the SPC Builder script.

The *constants* section of a CANDL file (typically starting from line 2) exported from a Petri net model with the same constants as depicted in Figure 6 would appear as follows (note that there may be other constants set):

```
1 constants:
2   valuesets[Main]
3 all:
4   int instances = 2;
5   int DX = 9;
6   int DY = 9;
```

The following adjustments are necessary to indicate to the builder which constants should be replaced with the users' parameters:

```
1 constants:
2   valuesets[Main]
3 all:
```

```
4 //dimensions
5 //instances
```

The location path of the modified CANDL file, including these adjustments, must be provided as a parameter when executing the SPC Builder section with parameters settings. The builder will then automatically replace the comments in lines 5 and 6 with:

```
1   valuesets[Main]
2 all:
3   int DX = <horizontal_grid_dimension_set_by_the_user>;
4   int DY = <vertical_grid_dimension_set_by_the-user>;
5   int instances = <number_of_instances_set_by_the_user>;
```

4.3 Writing SPC code

The SPC Builder simplifies the creation of SPC scripts by employing standard Python coding structures, such as loops, to eliminate the need for repeating similar lines of SPC code for the *onStep* object multiple times. Using this tool, each line of the conventional SPC code is typically written within an **.spc* file must be converted into a Python string. This approach allows repeated SPC lines to be condensed into a single line within a loop statement, or repeated sections to be stored once in a single variable and reused throughout the SPC script. The *BuilderSPC.py* file is the only file used for writing the SPC code as Python strings.

The *BuilderSPC* class consists of three functions: *onStepVariableDeclaration*, *onStepStepwise*, and *export*. Each of these methods corresponds to a common sub-property of the *Simulation Configuration* object in an SPC script. Specifically, *onStepVariableDeclaration* relates to the declaration section of the *onStep* property and should only include code for declaring variables, constants, and observers. *onStepStepwise* refers to the *stepwise* or *do* section of the *onStep* property and should contain only the conditions encoded as SPC code that Spike needs to check at each timestep of the simulation. *export* pertains to the *export* property used for defining which places, constants, and observers to export at the end of the simulation. The *BuilderSPC* object has a single variable declared within its constructor method, *self.o*. This variable contains an instance of the *BuilderOptions* class, which consists of a set of variables used for storing all the parameters reference to the parameter section prompted when running the builder. From the *BuilderSPC* object, you can access these options with *self.o*.

All three methods mentioned above include a variable called *spc*. This string variable is used to store SPC lines encoded as a Python string added within each method. These functions receive two parameters: *lns* and *pnt*.

lns contains the empty template of an SPC script as a string (defined in the *Constants.py* file), and its content will be saved in the auto-generated *.spc* script at the end of the *SPCBuilder* process. *pnt* is a line pointer that indicates to each method the specific line within the full SPC script as a string (*lns*) where the content of a particular method, stored in the *spc* variable, should be inserted.

4.3.1 onStepVariableDeclaration method

This method in the *SPCBuilder* represents the declaration section of the *onStep* property of an SPC script. The SPC code, written as a set of Python strings within this method, will be exported at the end of the generation

process into the default **.spc* file (named *spcOut.spc*).

When using the *SPCBuilder* in the context of the Hybrid Approach, this method should contain the lines of SPC code encoded as Python strings to declare all the variables, constants, and observers required for moving complex objects on the grid and interacting with the model, such as the initial x,y coordinates of each complex object.

By default, the code within this method is as follows:

```
    # Initialising the spc variable used to store
    # the SPC code as a Python string.
    spc = ""
    # self.o contains the options users set as
    # parameters on running the script
    # nco  is the number of complex obejcts set
    # by  the user
    nco = self.o.nco
    # 2D list containing the x,y location for each complex
    # object.
    # It uses the initial locations for each complex object
    # if set by the user, generate them randomly otherwise.
    # l0s  for 2 complex objects which x,y initial locations
    # are 3,2 and 4,1 does have the following content:
    # [[3,2], [4,1]].
    l0s = self.parseL0s() if self.o.l0 > 0 \
        else self.generateL0s()
    m = self
    # add comments in the SPC script.
    # m .n and m.t refer to self.n and self.n used for common
    # characters.
    # self.t(n) returns a string with n tabs characters.
    # eg. self.t(2) returns "\t\t".
    # This simplifies indentation when generating the spc file.
    # self .n return a string with a new line character.
    # eg. self.n() return "\n".
    spc += f'// {m.t(2)} ++++++++ Code generated from the Builder ++++++++  {m.
    spc += f'// ==================== ON STEP VARIABLE DECLARATION ==============
    # For each complex object.
    for i in range(nco):
        # Create  an id for the complex object.
        id = str(i+1)
        # Getting  the x,y initial location for the i_th
        # complex object.
        lx0 = l0s[i][0]
        ly0 = l0s[i][1]
        # Store  the x,y initial location lx0,ly0 for the i_th
        # complex object as a string. This will help in the next
        # lines to use these values for writing the SPC code
        # as a string.
```

```
41          x = f'X_{id}'
42          y = f'Y_{id}'
43          #  Writing   the SPC code for declaring the initial
44          # locations for the i_th complex object
45          # using Python string.
46          spc += f'{m.t(1)}{x} = {lx0} ; {m.n()}'
47          spc += f'{m.t(1)}{y} = {ly0} ; {m.n()}'
48          # Writing the SPC code for declaring the observers
49          # as Python strings
50          spc += f'{m.t(1)}{x}_obv:observe:{x};{m.n()}'
51          spc += f'{m.t(1)}{y}_obv:observe:{y};{m.n()}'
52      # write a comment
53      spc += f'//=================================================================
54      # lns   contains the full SPC script as a string.
55      # pnt   is a pointer indicating at which point of lns the
56      # content within the spc string has to be inserted.
57      #  Inserting   the spc content in lns at line pnt
58      lns.insert(pnt, spc)
```

This code snippet, when executed with the builder considering a 4x3 grid environment and 2 complex objects, will generate the following SPC code. This code is used for declaring the x, y variables and observers that define the initial locations of the two complex objects:

```
1   X_1 = 2 ;
2   Y_1 = 2 ;
3   X_1_obv:observe:X_1;
4   Y_1_obv:observe:Y_1;
5   X_2 = 1 ;
6   Y_2 = 2 ;
7   X_2_obv:observe:X_2;
8   Y_2_obv:observe:Y_2;
9 //=================================================================
10
11
12 def onStepStepwise(self, lns, pnt):
```

The *for loop* statement defined in line 32 of the *onStepVariableDeclaration* function serves two purposes: incrementing the counter that defines the *id* of each complex object, and writing the lines for declaring the location of each complex object (lines 4-7 of the SPC code) only once, rather than having to repeat these four lines for each complex object. Writing the code with the *SPCBuilder* allows for flexibility when changing the number of complex objects, grid dimensions, or other user-defined parameters without modifying the code.

Increasing or reducing the value of the number of complex objects (variable *nco* set in line 8) does not affect the code in the above method, due to the *for loop* statement in line 32, which writes the necessary variable dec-

larations according to the *nco* value set by the user. In contrast, manually writing the declaration directly into the SPC script would require the user to modify this SPC section each time a change in the number of complex objects is needed.

Furthermore, by setting the respective parameter when running the builder, users can specify the initial location of each complex object. If this parameter is not provided, the *SPCBuilder* will automatically generate the initial location for each complex object according to the provided grid dimensions.

4.3.2 onStepStepwise method

One of the key benefits of using Spike is its stepwise capability, which allows the model's state to be adapted for each timestep of the simulation based on pre-specified conditions set in the SPC's *onStep* property's do section. The Hybrid approach leverages this capability to enable complex objects to interact with the grid environment. However, manually writing the SPC code for the do section can be time-consuming, especially when dealing with a large number of conditions to check. The *onStepStepwise* method of the SPC builder simplifies this process by allowing you to write the SPC code for the do section as a Python string, as demonstrated in Section 4.3.1. This method provides access to all traditional programming language operators, making it easier to streamline the coding process.

Stepwise is crucial for this new approach because it allows the state of each complex object to be altered based on its location on the grid. Therefore, the purpose of the SPC code for the *do* section of an SPC script is to compare the x,y location of each complex object with all available positions on the grid and execute the SPC code inside the only conditions that will be true. The default code for the *onStepStepwise* method accomplishes this task by using a set of conditions to check each complex object's location and act accordingly.

```
    print(f'{self.log}writing onStep stepwise')
    # self.o contains the options users set as
    # parameters on running the script
    # nco  is the number of complex objects set
    # by  the user
    nco = self.o.nco
    # get the horizontal dimension of the grid
    # from  the option object
    x = self.o.x
    # get  the vertical dimensions of the grid
    # from  the option object
    y = self.o.y
    # Initialising the spc variable used to store
```

```
14      # the SPC code as a Python string. This variable
15      # will be appended to lns which is the full SPC
16      #  script  that will be exported to generate the
17      #  .spc script.
18      spc = ""
19      m = self
20      # for each complex object entity
21      # this  loop runs backwards for a better readability
22      # on generating the final .spc script
23      for i in range(nco-1, -1, -1):
24          #  Create  an id for the complex object. This id
25          # for each specific complex object has to be the same
26          # of  the one used on the onStepVariableDeclaration (line 63)
27          id = i+1
28          # for each available y coordinate within the grid
29          for _y in range(1, y+1):
30              # for each available x within the grid
31              for _x in range(1, x+1):
32                  #  create  all the conditions required to compare the
33                  # x,y location of the complex object 'id' with all
34                  # the available x,y positions of the grid.
35                  isIf = "if" if _x<=1 and _y<=1 else "else if"
36                  # if  the x position of the complex object 'id'
37                  # stored in the SPC variable X_'id' (declared in the
38                  # onStepVariableDeclaration function) is equal to '_x'
39                  # AND  the y position of the complex object 'id'
40                  #  store  in the SPC variable Y_'id' (declared in the
41                  # onStepVariableDeclaration function) is equal to '_y'
42                  spc += f'{m.t(1)}{isIf}(X_{id} == {_x} && Y_{id} == {_y}) {"{"}
43                  # if  the above condition is true (which means that
44                  # at a timestep t of the simulation the complex object
45                  # 'id' is in position '_x', '_y') execute the following code
46                  # (just some comments right now, but you should add the respect
47                  # actions)
48                  spc += f'{m.t(2)}// add here your code when complex object {id}
49                  spc += f'{m.t(2)}// is in location {_x};{_y}{m.n()}'
50                  #  close  the bracket of the if or else if condition
51                  spc += f'{m.t(1)}{"}"}{m.n()}'
52      # lns  contains the full SPC script as a string.
53      # pnt  is a pointer indicating at which point of lns the
54      # content within the spc string has to be inserted.
55      #  Inserting  the spc content in lns at line pnt
56      lns.insert(pnt, spc)
```

This code snippet, which assumes a 2x3 grid with 2 complex objects, generates the following SPC code that compares each complex object's x and y position with all available locations on the grid. The variables *X_n* and *Y_n*, declared in the *onStepVariableDeclaration* section (see Section 4.3.1),

specify the position of the complex object with ID *n*. The SPC code in line 43, encoded as a Python string, is converted to a series of *if* conditions in the final .spc script.

```
1
2  //LOOP_STATEMENT
3   if(X_2 == 1 && Y_2 == 1) {
4     // add here your code when complex object 2
5     // is in location 1;1
6   }
7   else if(X_2 == 2 && Y_2 == 1) {
8     // add here your code when complex object 2
9     // is in location 2;1
10  }
11  else if(X_2 == 1 && Y_2 == 2) {
12    // add here your code when complex object 2
13    // is in location 1;2
14  }
15  else if(X_2 == 2 && Y_2 == 2) {
16    // add here your code when complex object 2
17    // is in location 2;2
18  }
19  else if(X_2 == 1 && Y_2 == 3) {
20    // add here your code when complex object 2
21    // is in location 1;3
22  }
23  else if(X_2 == 2 && Y_2 == 3) {
24    // add here your code when complex object 2
25    // is in location 2;3
26  }
27  if(X_1 == 1 && Y_1 == 1) {
28    // add here your code when complex object 1
29    // is in location 1;1
30  }
31  else if(X_1 == 2 && Y_1 == 1) {
32    // add here your code when complex object 1
33    // is in location 2;1
34  }
35  else if(X_1 == 1 && Y_1 == 2) {
36    // add here your code when complex object 1
37    // is in location 1;2
38  }
39  else if(X_1 == 2 && Y_1 == 2) {
40    // add here your code when complex object 1
41    // is in location 2;2
42  }
43  else if(X_1 == 1 && Y_1 == 3) {
44    // add here your code when complex object 1
45    // is in location 1;3
```

```
  }
  else if(X_1 == 2 && Y_1 == 3) {
    // add here your code when complex object 1
    // is in location 2;3
  }
 }
```

For each timestep, t of the simulation, the conditions generated by the *onStepStepwise* method will be checked. Based on the current *X_n* and *Y_n* position of each complex object n at timestep t, only one of these conditions will be true and executed. The subsequent sections showcase examples of code that can be executed inside these conditions. The advantage of using this tool is that the user does not need to modify the Python code when changing the grid dimension, as it is set as an input parameter when running the script.

4.3.3 export method

This method within the BuilderSPC class is responsible for defining the contents of the *export* property of an SPC script, specifying which variables, constants, place markings, and observers should be exported as .csv files at the end of the simulation. Like the other methods discussed earlier, this method also encodes SPC code as Python strings to streamline the coding process. The default code within this method exports the *X_id_obv* and *Y_id_obv* observers that were defined in the *onStepVariableDeclaration* method.

```
    print(f'{self.log}writing spc export')
    m = self
    # m.o contains the options users set as
    # parameters on running the script
    # nco  is the number of complex objects set
    # by  the user
    nco = self.o.nco
    # Initialising the spc variable used to store
    # the SPC code as a Python string. This variable
    # will be appended to lns which is the full SPC
    # script  that will be exported to generate the
    #  .spc script.
    spc = ""
    #  exporting  all the places
    spc += f'{m.t(1)}places: [];{m.n()}'
    #  exporting  the observers
    exb = f'{m.t(1)}observers: ['
    # for  each complex object
    #  export  its associated observer decleared
    # in  on step variabel declaration
    for i in range(nco):
```

```
22          id = i+1
23          exb += f'X_{id}_obv, Y_{id}_obv'
24          # add  , to separate observers
25          exb += "" if i == nco-1 else ","
26      spc += f'{exb}];{m.n()}'
27      # lns  contains the full SPC script as a string.
28      # pnt  is a pointer indicating at which point of lns the
29      # content within the spc string has to be inserted.
30      #  Inserting  the spc content in lns at line pnt
31      lns.insert(pnt, spc)
```

The SPC code inserted as Python strings in lines 16, 18, 24, 26, and 27 generates the following results (line 3, and 4):

```
1  //PLACE_EXPORT
2    places: [];
3    observers: [X_1_obv, Y_1_obv,X_2_obv, Y_2_obv];
4
5    csv: {
6        \sep: ",";// Separator
7        file: "result"
8        \<< ".csv";// File name
9    }
10 }
```

4.4 Executing the builder

The SPC builder is a Python script that needs to be executed from the command line once all the requirements outlined in Section 2 have been met, and the steps described in Sections 4.1 and 4.2 have been completed. To get started, navigate to the SPC-python-builder folder from your terminal using the command *cd*:

```
1  $ cd SPC-python-builder
```

The builder requires the following mandatory parameters, if not provided the default values will be considered:

1. x: defines the x dimension for the grid environment. As default, it is set to 3.

2. y: defines the y dimension for the grid environment. As default, it is set to 3.

3. o: defines the number of instances for a Petri Net colour (complex object). As default, it is set to 2

4. c: the location path of the Petri Net CANDL file with the changes illustrated in Section 4.2

x, y, o are the parameters that the builder will use to overwrite the constants within the CANDL file as described in Section 4.2. You can run the script by typing:

```
$ python3 builder.py --x=x_grid_environment \
--y=y_grid_environment \
--o=no_complex_obejcts \
--c=candl_path
```

For example:

```
$ python3 builder.py --x=5 --y=6 --o=3 --c=./model.candl
```

5 Examples

The SPC builder was developed to simplify the process of creating SPC scripts for implementing the Hybrid approach in Petri net models that involve dynamic entities. To demonstrate the need for this approach and the use of the SPC builder to support its implementation, we have selected three models: Whale feeding behaviour, Slime mould Dictyostelium clumping behaviour, and Quorum sensing and biofilm production. The sections below showcase how the SPC builder has been utilized to implement these three dynamic systems, which require the interaction of complex objects (Whale, Dictyostelium, and Biofilm) with a grid environment of varying shapes and sizes. For a more detailed explanation of the biological background and theoretical aspects of these models, please refer to our report (insert reference). All the files showcased in the next sections are available at link with the repository with all the files

5.1 Whale feeding behaviour

5.1.1 Introduction to the Whale feeding scenario

This section aims to explain how to prepare the SPC script using the SPC builder for a simple model that leverages the Hybrid mechanism to allow the movement of complex entities within a grid environment and their interaction with atomic objects. The objective is to simulate the behaviour of a whale, with a focus on its movement and feeding behaviour, and observe its response to different environmental conditions. To achieve this, a dynamic Petri net model needs to be designed, which incorporates the whale's internal biological structure and an abstract representation of phytoplankton as a source of food. The implementation of this scenario using the Hybrid approach requires the incorporation of a Petri net to manage the model's physical structure and the grid-based movement of the atomic objects, the plankton. Additionally, a stepwise mechanism will be employed to enable the movement and feeding behaviour of the complex object, the whale. The SPC builder streamlines the process of writing the SPC script that encodes the stepwise mechanism needed to simulate the whale's movements and feeding behaviour. The effectiveness of the builder is noticeable when the model requires SPC code for handling several whales located on a large grid. In this scenario, the Petri net model is used for:

- encoding the space into a coloured place defining an X*Y Grid environment (in this example a 3x1 grid)

- moving the atomic objects across the grid
- modelling the internal structure of the whale that will be moved atomically by the stepwise mechanism

The Stepwise mechanism is used to encode the following actions:

- storing the whales' positions (using 2 variables X and Y for each whale)
- moving the whales across the grid environment defined within the Petri net model
- feeding the whale by "eating" the food located in the Grid environment

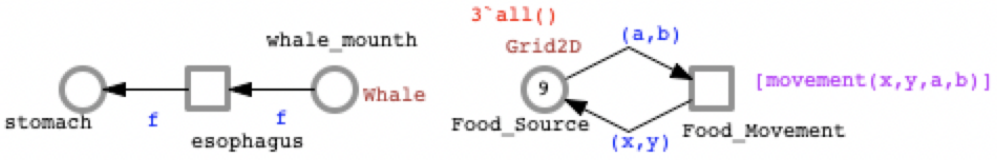


Figure 7: This case study focuses on a simple model consisting of two parts: the Whale and its internal structure, and the environment and Plankton movement. The Whale, modelled as a Petri net colour set "Whale" contains two places ("whale_mouth" and "stomach") and one transition ("esophagus"). For demonstration purposes, the Whale is modelled from a high-level perspective to showcase the Hybrid approach's ability to handle Complex Objects. The environment is represented by a grid, where each grid position is represented as an X, Y tuple. The shape and dimensions of the environment are defined through a product formula in the Petri net colour set "Grid2D" on the place "Food_Source". In this example, the environment is a simple 3x1 grid, as shown in the uncoloured version in Figure 8 of the ANDL version of the model. The place markings on "Food_Source" represent the number of Plankton (atomic objects) in each position. At Time 0, each position on the grid (3_1, 3_2, 3_3) contains three Plankton. The feeding behaviour of the Whale begins in the "whale_mouth" place, and food travels from the Whale's mouth to its stomach via the "esophagus" transition. The SPC script encodes the connection between the Whale's mouth and the environment and the transfer of food from the environment to the Whale's mouth. The atomic movement of the entire Whale structure, consisting of two places and one transition, is encoded in the Stepwise system, as shown in the SPC script below.

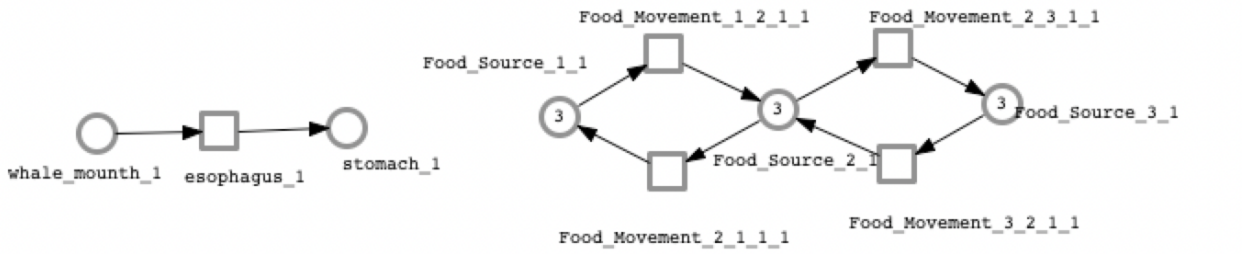


Figure 8: The model depicted in Figure 7 can be represented in ANDL format, providing a concise and clear visualization of its components. The environment is modelled as three places, "Food_Source_1_1", "Food_Source_2_1", and "Food_Source_3_1", encoding the 3x1 grid structure. Meanwhile, the internal anatomy of a whale, including its two compartments (stomach_1 and whale_mouth_1) and the transition (esophagus_1), is represented by a Petri net colour code-named "Whale". In the ANDL format, each instance of a "Whale" has a unique integer ID ranging from 1 to n, where n is the total number of whale instances (in this case, only 1 instance is considered). All places and transitions belonging to the same whale instance have the same ID in the ANDL file. This ensures that the variables used to store the X and Y coordinates in the SPC code, which are used to identify the location of the whale, have consistent ID values within their names.

The SPC script presented below implements the stepwise mechanism that enables the movement and feeding behaviour of the whale, which is part of the use case model. The position of objects in a grid-based environment is determined by an X and Y tuple, and to track the whale's position, two variables, whale_1_X_obv and whale_1_Y_obv (declared in lines 36 and 37), are used to store the current coordinates of the whale instance. These coordinates are used to identify the position of the whale within the grid space. In the Hybrid approach, to move a complex object atomically, the positions of all the places and transitions included in its internal structure must change simultaneously. For instance, to move the whale, the positions of the two places (whale_mouth and stomach) and the transition (esophagus) must be changed in unison to a new location. To enable a complex object to interact with its grid environment, the stepwise code compares the object's position, stored as X and Y coordinates, with all the possible X, and Y locations available in the grid. Only one of these possible conditions will be true, and it identifies the block of SPC code to be executed when the complex object is located at a specific grid position. In this case study, the interaction between the complex object (whale) and the grid environment mimics the interaction between a whale and a food source. The SPC code transfers the food source from a grid location to the whale_mouth place of the whale instance/s located at the same position, effectively representing the feeding behaviour. The interaction is implemented in lines 47-49 to transfer the food source located in position 1;1 of the grid environment to the whale in position 1;1, and in the same way for other available positions in lines 58-60 and 65-67. The stepwise mechanism also enables the whale to move atomically across the 3x1 grid environment. The starting position of the whale is 1,1, but by modifying this value with available grid positions, the whale's movement can be triggered. For example, in line 72, the value of whale_1_X_obv is increased by one unit, resulting in the whale moving from its starting position to the right.

```
1  import: {
2    from: "whaleExampleModel.andl";
3  }
4  configuration: {
5    model: {
6      places: {   }
7    }
8  simulation:
9  {
10
11   name: "SIR";
12
```

```
type:continuous : {
  solver:  BDF: {
    semantic: "adapt";
    iniStep: 0.1;
    linSolver: "CVDense";
    relTol: 1e-5;
    absTol: 1.0e-10;
    autoStepSize: false;
    reductResultingODE: true;
    checkNegativeVal: false;
    outputNoiseVal: false;

  }
  single: true;
}

interval: 0:50:50;

/*
* Stepwise simulation
*/
onStep: {
  /* Variable declaration */
  Whale_1_X : 1;
  Whale_1_Y : 1;
  whale_1_X_obvs:observe:Whale_1_X; // observer of X
  whale_1_Y_obvs:observe:Whale_1_Y; // observer of Y

  do: {

    /* If the whale is in position 1;1 */
    if(whale_1_X_obvs == 1 && whale_1_Y_obvs == 1) {
      /* If the whale is in position 1;1 eats the food in
       *  Food_Source_1_1 */
      place.whale_mounth_1 = place.whale_mounth_1 +
                             place.Food_Source_1_1;
      place.Food_Source_1_1 = 0;
      /* Changing the X value from 1 to 2 enables the
      * movement of the whale on the X axes as well as
      * moving the whale into the nexT
      * location on the right*/
    }
    /* If the whale is in position 2;1 eats the food in
    *  Food_Source_2_1 */
    else if (whale_1_X_obvs == 2 && whale_1_Y_obvs == 1) {
      place.whale_mounth_1 = place.whale_mounth_1 +
                               place.Food_Source_2_1;
      place.Food_Source_2_1 =0;
    }
```

```
62        /* If the whale is in position 3;1 eats the food in
63        *  Food_Source_3_1 */
64        else if (whale_1_X_obvs == 3 && whale_1_Y_obvs == 1) {
65          place.whale_mounth_1 = place.whale_mounth_1 +
66                                       place.Food_Source_3_1;
67          place.Food_Source_3_1 = 0;
68        }
69                  // move Whale 1 atomically on the right until
70                  // there is some space
71              if (whale_1_X_obv < 3) {
72                   whale_1_X_obv = whale_1_X_obv + 1;
73              }
74
75      }
76    }
77    /* Exporting places and observers values */
78    export: {
79      places: ["Food_Source_._.","stomach_.","whale_mounth_."];
80      observers: ["whale_1_X_obvs","whale_1_Y_obvs"];
81      csv: {
82        sep: ",";// Separator
83          file: "result"
```

The code block between lines 44 to 70 needs to be replicated for each colour of the "Whale" colour set. If the model has many whales, this manual repetition process can be time-consuming. To streamline this process, the following sub-section explains how the SPC builder is used.

5.1.2 SPC builder for the Whale feeding scenario

The above SPC script demonstrates how stepwise logic is utilised to implement the Hybrid approach for the interaction of complex objects (whales) and atomic entities (grid locations and phytoplankton) to simulate the movement and feeding behaviours of a single whale in a 3x1 grid environment. However, as the number of whale instances or the dimensions of the grid increases, the manual writing of stepwise instructions becomes a time-consuming and challenging task. This section describes how the SPC builder has been configured with the Petri net model to automatically generate repeated sections of SPC code for two whales in a 3x2 grid environment (as shown in Figure 9) using traditional Python computer language operators. As discussed in Section 4, the implementation of the Hybrid approach using the SPC builder involves setting up the Petri net model, configuring CANDL, and writing the SPC code. The following three subsections provide details on how these steps have been completed for the whale feeding model.

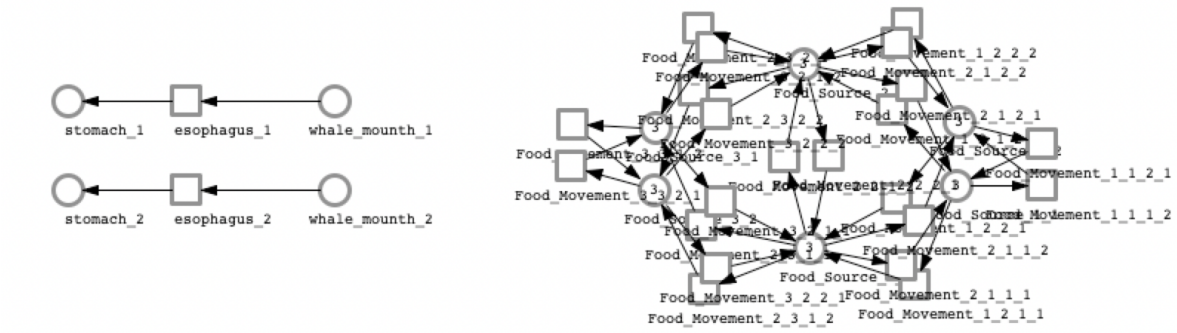


Figure 9: Unfolded Petri net models with a 3x2 grid and two whales.

5.1.3 Setting up the Petri net model

In Section 4.1, we explained how to set up a Petri net model for integrating the Hybrid approach using the SPC builder. In the case of the Whale Feeding model (Section 7), the colour sets for defining the grid environment and the complex object (whale) are configured as per the convention mentioned in Section 4.1. There are three Simple Color Sets required for this model: X, Y, and *Whale* (as shown in Figure 11). X and Y are of integer type and used to specify the grid dimensions. These two Simple Color Sets are used for defining the Compound Color Set *Grid2D* (as shown in Figure 12), derived from the product of X and Y. *Grid2D* is used to shape the environment. The values of these three Simple Color Sets (X, Y, and *Whale*) are declared by the model Constants DX, DY, and *instances* (as shown in Figure 10). For example, in the case of the model presented above (2 whales and a 3x2 grid environment), the constant configuration is as follows. By changing the value of these constants, the environment's dimensions or shape can be changed, or the number of whales can be increased or decreased. The SPC builder writes these constants using its internal procedure, setting each of them with the respective parameter the user sets when running the builder. It is important to ensure that the Simple Color Sets use the exact same constants to define their values so that they match the constant names set by the builder with the values the user has set as parameters. Once the model is built and the colour sets are defined, the model can be exported as CANDL.

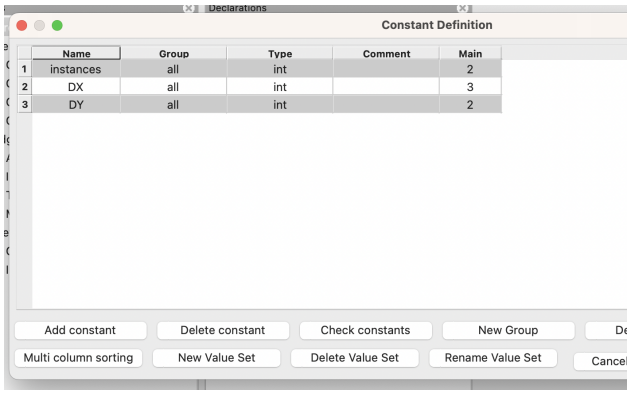


Figure 10: Constants are used for defining the dimension of the grid and the number of whales. These constants will be overwritten from the SPC builder by changing the CANDL file given as input on running the builder itself.

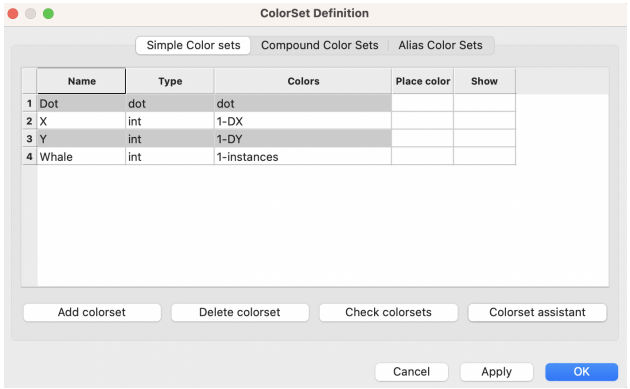


Figure 11: Simple colour set definition

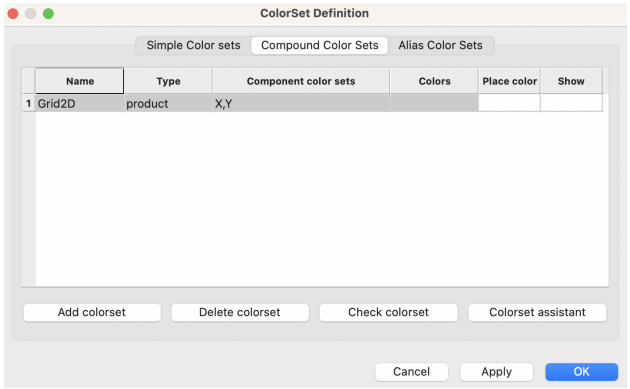


Figure 12: Compound colour set definition

5.1.4 Exporting and configuring CANDL

For the SPC builder to generate the ANDL and SPC script in a semi-automated way, the CANDL file of the model needs to be provided as input. The process of exporting the model into a CANDL file differs depending on the environment used for designing the Petri net model. For instance, in Snoopy, the model can be exported by selecting File, Export, and "Export to CANDL using dssd_util". Before running the builder with the exported CANDL file, some minor modifications are required as explained in Section 4.2. By default, the exported CANDL file for the model with one whale and a 3x1 grid environment will look like this:

```
  valuesets[Main]
all:
  int instances = 2;
  int DX = 3;
  int DY = 2;

colorsets:
  Dot = {dot};
  X = {1..DX};
  Y = {1..DY};
  Whale = {1..instances};
  Grid2D = PROD(X,Y);
```

The required changes are to indicate the lines in the CANDL that the builder will use to write the model's constants with the user's parameters. These constants will be used to define the values for the three simple colour sets presented in the section above (Section 5.1.3). To make these changes, the lines within the *Constants* section of the exported CANDL that define the constants for the Simple Color Sets (*DX*, *DY*, *instances*) have to be replaced with specific comments, as shown below:

```
  valuesets[Main]

all:
//dimensions
//instances

colorsets:
  Dot = {dot};
  X = {1..DX};
  Y = {1..DY};
  Whale = {1..instances};
  Grid2D = PROD(X,Y);
```

These comments are customizable in *Constants.py*. The CANDL is now ready to be used as input for the SPC builder.

5.1.5 SPC for the Whale feeding model

In order to model the movement of the whale and its interaction with the grid-based environment, the stepwise system will be encoded in the SPC builder. This system allows for the atomic movement of the whales and their internal structure, as well as the consumption of food sources located in the same grid position. As the grid becomes larger and the number of whales increases, manually writing the necessary SPC sections becomes more time-consuming. To streamline the process, the SPC builder can be used to automatically generate these sections. To begin, the position of each whale must be declared in the SPC code as two values: one for the horizontal coordinate and one for the vertical coordinate. The *onStepVariableDeclaration* method in the SPC builder is responsible for containing the SPC code written as a Python string used to declare all the variables, constants, and observers. For example, to encode the position of two whales in a 3x2 grid environment, the following lines could be added to the *onStepVariableDeclaration* method:

```
    # Initialising the spc variable used to store
    # the SPC code as a Python string.
    spc = ""
    # self.o contains the options users set as
    # parameters on running the script
    # nco  is the number of complex objects set
    # by  the user
    nco = self.o.nco
    # 2D list containing the x,y location for each whale
    # It uses the initial locations for each whale object
    # if set by the user, generate them randomly otherwise.
    l0s = self.parseL0s() if self.o.l0 > 0 else self.generateL0s()
    m = self
    # add comments in the SPC script.
    # m .n and m.t refer to self.n and self.n used for common
    # characters.
    # self.t(n) returns a string with n tabs characters.
    # eg. self.t(2) returns "\t\t".
    # This simplifies indentation when generating the spc file.
    # self .n return a string with a new line character.
    # eg. self.n() return "\n".
    spc += f'// {m.t(2)} ++++++++ Code generated from the Builder ++++++++  {m.
    spc += f'// ==================== ON STEP VARIABLE DECLARATION ==============
    # For each instance of the whale
    for i in range(nco):
        # Create an id for the whale
        id = str(i+1)
        #  Getting  the x,y initial location for the i_th
        # whale store in l0s variable.
        lx0 = l0s[i][0]
```

```
        ly0 = l0s[i][1]
        #  Writing  the SPC code for declaring the initial
        # locations for the i_th whale
        # using Python string.
        # The  x value for the whale i_th
        # will  be Whale_<i_th>_X
        # The  y value for the whale i_th
        # will be Whale_<i_th>_Y
        x = f'Whale_{id}_X'
        y = f'Whale_{id}_Y'
        spc += f'{m.t(1)}{x} : {lx0} ; {m.n()}'
        spc += f'{m.t(1)}{y} : {ly0} ; {m.n()}'
        # Writing the SPC code for declaring the observers
        # as Python strings
        spc += f'{m.t(1)}{x}_obv:observe:{x};{m.n()}'
        spc += f'{m.t(1)}{y}_obv:observe:{y};{m.n()}'
    # Write a comment
    spc += f'//=================================================================
    lns.insert(pnt, spc)
//      ++++++++ Code generated from the Builder ++++++++
```

The SPC builder converts this Python string to SPC code using the logic described in Section 4.3.

```
  Whale_1_X : 1 ;
  Whale_1_Y : 1 ;
  Whale_1_X_obv:observe:Whale_1_X;
  Whale_1_Y_obv:observe:Whale_1_Y;
  Whale_2_X : 1 ;
  Whale_2_Y : 1 ;
  Whale_2_X_obv:observe:Whale_2_X;
  Whale_2_Y_obv:observe:Whale_2_Y;
//=====================================================================
```

These two variables represent the positions of each whale over the simulation time (at timestep 0 Whale 1 is located at position 1,1 and Whale 2 at position 2,1).
The central logic of all Hybrid approaches lies within the SPC code section named *onStep stepwise*. For the Whale Feeding model, stepwise is crucial to enable the whales to move on the grid and consume phytoplankton. The movement of the whales is implemented by modifying the variables that indicate their positions on the grid. The movement depends on the rules defined for changing the location, and in this model, the only permitted movement is one unit to the right at each timestep. Therefore, for each iteration of the simulation, the SPC code increases the *Whale_X_ID* variable of each whale by one unit. For the feeding behaviour, the stepwise system transfers the phytoplankton located in the same grid position as the whale

to the whale's mouth place. This feeding behaviour is encoded separately for each whale instance.

```
    print(f'{self.log}writing onStep stepwise')
    nco = self.o.nco
    x = self.o.x
    y = self.o.y
    spc = ""
    m = self
    # for each complex object entity
    for i in range(nco-1, -1, -1):
        # Get  the id of the whale
        id = i+1
        # if there is space on the right-hand side
        # move  the whale_x on the right by increasing
        # the  value of Whale_id_X_obv of one unit
        spc += f'{m.t(1)}if (Whale_{id}_X_obv < {x}) {"{"}{m.n()}'
        spc += f'{m.t(2)} Whale_{id}_X_obv = Whale_{id}_X_obv + 1; {m.n()}'
        spc += f'{m.t(1)}{"}"}{m.n()}'

        # for each available vertical value within
        # the grid environment
        for _y in range(1, y+1):
            # for each available horizontal value within
            # the grid environment
            # write the behaviour for each whale
            for _x in range(1, x+1):
                # this  is used just as a flag to write "if"
                # or "else" in the final SPC
                isForIf = "if" if _x<=1 and _y<=1 else "else if"
                # if  Whale_id_X;Whale_id_Y is equal to _x;_y which
                # means when Whale_<id> is located in the
                # grid position _x;_y
                spc += f'{m.t(1)}{isForIf}(Whale_{id}_X == {_x} && Whale_{id}_Y
                # then  get the marking value of the food
                # source in the grid place _x;_y
                # (place.Food_Source_x_y) as well
                # as the same grid position where
                # the whale <id> is currently located
                # and move it to the place of the whale
                #  <id> mouth.
                # This  exchange of markings (tokens)
                # encodes the feeding behaviour
                spc += f'{m.t(2)}place.whale_mounth_{id} = place.whale_mounth_
                # once  all the markings have been moved
                # from the place.Food_Source_x_y
                # to the whale <id> month the marking
                # of the place source
                # can be empty as well as set to 0
```

```
                spc += f'{m.t(2)}place.Food_Source_{_x}_{_y} = 0;{m.n()}'
                spc += f'{m.t(1)}{"}"}{m.n()}'
    lns.insert(pnt, spc)
```

Considering two whales, this Python code will generate the following SPC code:

```
     Whale_2_X_obv = Whale_2_X_obv + 1;
}
if(Whale_2_X == 1 && Whale_2_Y == 1) {
    place.whale_mounth_2 = place.whale_mounth_2 + place.Food_Source_1_1;
    place.Food_Source_1_1 = 0;
}
else if(Whale_2_X == 2 && Whale_2_Y == 1) {
    place.whale_mounth_2 = place.whale_mounth_2 + place.Food_Source_2_1;
    place.Food_Source_2_1 = 0;
}
else if(Whale_2_X == 3 && Whale_2_Y == 1) {
    place.whale_mounth_2 = place.whale_mounth_2 + place.Food_Source_3_1;
    place.Food_Source_3_1 = 0;
}
else if(Whale_2_X == 1 && Whale_2_Y == 2) {
    place.whale_mounth_2 = place.whale_mounth_2 + place.Food_Source_1_2;
    place.Food_Source_1_2 = 0;
}
else if(Whale_2_X == 2 && Whale_2_Y == 2) {
    place.whale_mounth_2 = place.whale_mounth_2 + place.Food_Source_2_2;
    place.Food_Source_2_2 = 0;
}
else if(Whale_2_X == 3 && Whale_2_Y == 2) {
    place.whale_mounth_2 = place.whale_mounth_2 + place.Food_Source_3_2;
    place.Food_Source_3_2 = 0;
}
if (Whale_1_X_obv < 3) {
     Whale_1_X_obv = Whale_1_X_obv + 1;
}
if(Whale_1_X == 1 && Whale_1_Y == 1) {
    place.whale_mounth_1 = place.whale_mounth_1 + place.Food_Source_1_1;
    place.Food_Source_1_1 = 0;
}
else if(Whale_1_X == 2 && Whale_1_Y == 1) {
    place.whale_mounth_1 = place.whale_mounth_1 + place.Food_Source_2_1;
    place.Food_Source_2_1 = 0;
}
else if(Whale_1_X == 3 && Whale_1_Y == 1) {
    place.whale_mounth_1 = place.whale_mounth_1 + place.Food_Source_3_1;
    place.Food_Source_3_1 = 0;
}
else if(Whale_1_X == 1 && Whale_1_Y == 2) {
    place.whale_mounth_1 = place.whale_mounth_1 + place.Food_Source_1_2;
```

```
44      place.Food_Source_1_2 = 0;
45  }
46  else if(Whale_1_X == 2 && Whale_1_Y == 2) {
47      place.whale_mounth_1 = place.whale_mounth_1 + place.Food_Source_2_2;
48      place.Food_Source_2_2 = 0;
49  }
50  else if(Whale_1_X == 3 && Whale_1_Y == 2) {
51      place.whale_mounth_1 = place.whale_mounth_1 + place.Food_Source_3_2;
52      place.Food_Source_3_2 = 0;
53  }
```

Lines 1-3 and 28-30 of the SPC code move Whale 1 and Whale 2 atomically to the next available right grid position. Other "if" conditions are used to determine the behaviour of each whale when it is located in a specific grid position. Since the grid environment has a size of 3x2, there is a condition for each available position on the grid that must be checked for each whale's location at each simulation timestep. These positions include [1,1], [2,1], [3,1], [1,2], [2,2], and [3,2].

5.2 Export

The following Python code contains the *export* method that the SPC Builder requires to list all the places, constants, observers, and variables that the user wants to export as a CSV file at the end of the simulation. This method is responsible for exporting the simulation data to an external file that can be analysed using external tools. The method has to be set with the correct parameters for the user's specific model, such as the filename and the set of variables and constants that the user wants to include in the exported data.

```
1       print(f'{self.log}writing spc export')
2       m = self
3       # m.o contains the options users set as
4       # parameters on running the script
5       # nco  is the number of whales set
6       # by  the user
7       nco = self.o.nco
8       # Initialising the spc variable used to store
9       # the SPC code as a Python string. This variable
10      # will be appended to lns which is the full SPC
11      #  script  that will be exported to generate the
12      #  .spc script.
13      spc = ""
14      #  exporting  all the places
15      spc += f'{m.t(1)}places: [];{m.n()}'
16      #  exporting  the observers
17      exb = f'{m.t(1)}observers: ['
18      # for  each whale
```

```
    # export  its associated observer decleared
    # in  on step variable declaration
    for i in range(nco):
        id = i+1
        exb += f'Whale_X_{id}_obv, Whale_Y_{id}_obv'
        # add  , to separate observers
        exb += "" if i == nco-1 else ","
    spc += f'{exb}];{m.n()}'
    # lns  contains the full SPC script as a string.
    # pnt  is a pointer indicating at which point of lns the
    # content within the spc string has to be inserted.
    #  Inserting  the spc content in lns at line pnt
    lns.insert(pnt, spc)
```

These Python lines will be then converted by the builder into the following SPC code and allow the export of the observers used for tracking the position of each whale.

```
//PLACE_EXPORT
 places: [];
 observers: ["Whale_1_X_obv", "Whale_1_Y_obv","Whale_2_X_obv", "Whale_2_Y_obv"

csv: {
sep: ",";// Separator
file: "result"
<< ".csv";// File name
}
}
}
```

The final csv file sets with this export definition will contain all the markings'values of each place of the Petri net model and the history of locations of the whales at each timestep of the simulation.

5.3 Using the builder for the Whale model

The command to run the SPC builder to generate the ANDL and SPC scripts required for running the Hybrid Approach on the Whale model consisting in two whales on a 3x2 grid environment is the following:

```
$ python3 main.py --x=3 --y=2 --o=2 --f=whaleExampleModel.candl
```

The CANDL file (whaleExampleModel.candl), as well as all the files and scripts required for reproducing this model, can be found in the *Python-Builder-Whale-Example* folder at link with the file.

5.4 Dictyostelium

Dictyostelium (Dicty) is considered to be a social amoeba because these unicellular microorganisms communicate with one another if suitably close. Under normal circumstances, when there is no environmental pressure, Dicty moves around randomly in a diffusion-like behaviour [EMRKG17, VHG19]. But under what? pressure or starvation they gather together and create a multicellular aggregated society to survive. In the Dictyostelium (Dicty) model, the cells are attracted towards the concentrations of the signalling molecules which are being diffused on the grid. This means that the attraction is not in one place and does not show a stable gradient. As a result, the bacteria gather around the "Wall" while the Dictyostelium cells might gather anywhere on the grid as long as there is a high concentration of cAMP. The model can be observed in Figure 13. This abstract model demonstrates the movement of Dictyostelium towards cAMP, while cAMP is diffused on the grid. It is important to note that Dictyostelium do not absorb the molecules and just senses them in order to move towards it. Once located on the grid, the cells start producing cAMP and naturally will move towards where the concentration of this molecule is the highest. The Degradation transition is only connected to the cAMP in order to prevent too much concentration of the molecule on the grid. Otherwise, there will be too much cAMP everywhere and the movement would not happen. The implementation of this biological system required the use of Hybrid Approach capabilities to enable Dicty objects (with all their internal structure) to move (sense) towards the cAMP. As illustrated previously the Hybrid approach is partially implemented using the Petri net model, and partially using the Stepwise SPC mechanism. This is a suitable system requiring the Hybrid approach where the model, as well as the internal structure of the Dicty (complex object) and the cAMP (atomic object) diffusion mechanism, are implemented using the traditional coloured Petri net model, and the atomic movement of the Dictys toward those localities in the grid with highest cAMP leveraging the Stepwise SPC system. This scenario has been prototyped in Figure 13.

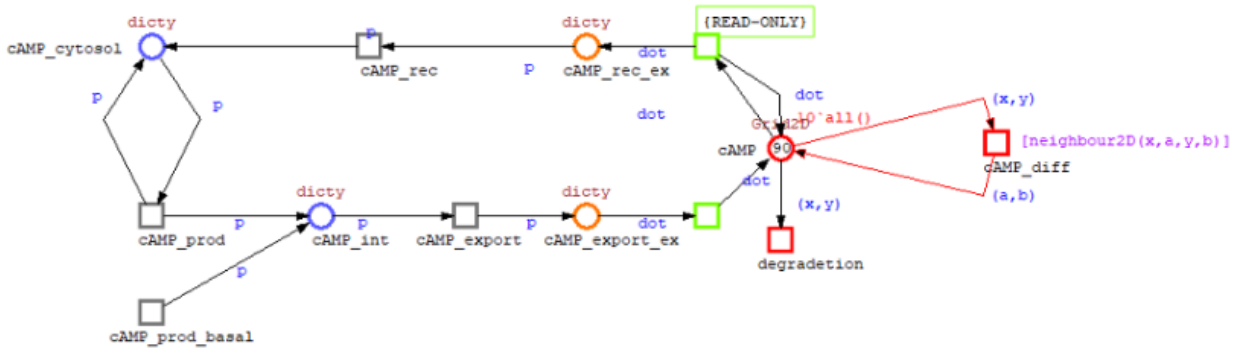


Figure 13: Abstract model for the Dicty system. This model is a prototype illustrating how the Dicty system can be designed using the Hybrid Approach. It can be split in three sub-sections: Dicty with internal structure (left side), SPC stepwise layer (green transitions in the middle), and cAMP diffusing on the grid (red section on the right-hand side). The Hybrid approach is needed because the Dicty object with all its internal places and transitions needs to sense and so interact with cAMPs which are atomic objects moving in the grid for moving towards these localities with the highest cAMP gradient. The interaction between a complex object and an atomic entity cannot be modelled using the traditional Petri due to the lack of specific operators. The Hybrid Approach solve this problem by integrating the SPC Stepwise tool (green transitions) as the operator required to support this interaction. This is just a case study showcasing the advantages of this new approach which can be used in all these models requiring complex objects to communicate and interact with atomic entities sharing the same space.

6 Acknowledgements

This research was funded by the Leverhulme Trust under their Emeritus Fellowship scheme awarded to David Gilbert, Project number EM-202-0-086\9, (August 2021 to September 2023). The development work was done by Francesco Rinaldi (undergraduate student research assistant), who also wrote this practical manual. Thanks are due to Professor Monika Heiner of Brandenberg Technical University who provided the simulator platforms Spike and Snoopy, part of the Petrinuts software platform, and also to Dr. Leila Ghanbar of Brunel University London whose earlier doctoral work laid the foundations for this work.