Using Z in the development and maintenance of computational models of real-world systems

Shahrzad Moeiniyan Bagheri[1,2], Graeme Smith[1], and Jim Hanan[2]

[1] School of Information Technology and Electrical Engineering
The University of Queensland, Australia
[2] Queensland Alliance for Agriculture and Food Innovation
The University of Queensland, Australia

Abstract. There are two main challenges in developing computational models of a real-world phenomena. One is the difficulty in ensuring clear communication between the scientists, who are the end-users of the model, and the model developers. This results from the difference in their backgrounds and terminologies. Another challenge for the developers is to ensure that the resultant software satisfies all the requirements accurately. Utilising a formal notation such as Z which is easy to learn, read, understand and remember can address these issues by (a) acting as a means to unambiguously communicate between scientists and simulation developers, and (b) providing a basis for systematically producing and maintaining simulation code that meets the specification. In this paper, we describe a translation scheme for producing code for the widely used agent-based simulation environment NetLogo from Z specifications. Additionally, we report on the use of the approach on a real project studying the movement of *chyme*, i.e. food undergoing digestion, through a pig's intestine as a means of understanding the effect of dietary fibre on human health.

1 Introduction

Studying real-world processes through experimental observation can be technically difficult. It can also be costly both in time and resources, and in certain areas ethical issues can arise. A more convenient approach is to develop a computational model of the system. Such a model allows scientists to uncover patterns in the studied system and to determine the system parameters and factors that are the most influential. While the visualisations provided by a computational model allows scientists to observe macro-level behaviour, this behaviour is only representative of their understanding of how the system works when that understanding has been accurately encoded. Furthermore, such visualisations represent specific system behaviours, not the general system behaviour. Ensuring an accurate encoding of the general behaviour can be difficult to achieve, particularly when the developers of the model are not part of the scientific team and are from different backgrounds. The situation can become even worse if developers and scientists have different and even sometimes conflicting terminologies. For example, they may use different terms for the same concepts, or a single term for different concepts.

An additional problem is that the scientists' understanding of the system may evolve over time and the model needs to be modified to reflect this. Again it is important that the modifications are an accurate reflection of the required change. This is facilitated if the model is in terms of constructs that are easy to learn and remember. The constructs of the implementation language of a typical simulation environment are too low-level (i.e., too close to programming constructs) to satisfy this criteria for scientists who are not familiar with programming.

To overcome these issues, two considerations are essential. Firstly, the system needs to be specified and the design ideas and decisions documented using a method that is easy to learn, read, understand and remember by all the people involved, both during the development and for future purposes like testing and maintenance. Such a method should make the communication between the scientists and developers, and also between the development team members, more convenient, efficient and well-documented. Secondly, the applied method for specifying the system should provide a basis for ensuring efficient and accurate implementation of the specified requirements. Not only should this facilitate the production of the code, but also its maintenance. The changes in the scientists' understanding of the system, and as a result in the specification, should be readily incorporated into the code. This reduces the burden of the resultant model's integrity assurance and its maintenance from the developers' shoulders.

To achieve these two goals, formal methods can be utilised. Formal specifications provide a communication method that does not include the ambiguities that are found in informal (e.g., natural language) specifications [3]. These methods also allow the developers to gain a clear understanding of the system, before starting the implementation process. Moreover, such specifications can form the basis of a systematic approach for deriving simulation code. This is in contrast to semi-formal specifications (e.g., UML [2]) that only achieve the first goal.

Recently, we developed simulation software for biologists at the Centre for Nutrition and Food Sciences (CNAFS), The University of Queensland, who are examining the effect of dietary fibre on human health. For facilitating the communication in the development process and future maintenance of the software, we utilised the formal notation Z [10, 13]. Z was chosen due to it being a simple extension of set theory and first-order predicate logic which is relatively easy to learn. Since the biologists are not necessarily familiar with programming, notations supporting program-like constructs (such as B [1]) or program-like structuring (such as Object-Z [9]) were seen as having more concepts to learn, and hence being less suitable. It should be noted that in order to achieve the goal of having a fairly simple and easy to learn notation, the use of Z in this research was restricted to avoid more complex notations and certain modelling techniques, such as promotion [13], which are regarded as difficult to understand for non-computer scientists.

For the purpose of implementing this simulator, the NetLogo [11] simulation environment was chosen. NetLogo is a modelling language and environment that is widely used for developing agent-based simulation software. The agent-based approach considers smaller components of the system as autonomous entities (agents) that can form more complex system-level behaviours while performing relatively simple interactions with each other and with their environment [8]. Additionally, a systematic translation scheme from Z to NetLogo code was defined. This made the implementation much easier and enabled us to ensure that the simulation software behaved as required and specified.

Mens and Van Gorp [6] argue that even source code can act as the specification of a system being studied. As a result, the need for utilising Z for specifying the system requirements might be questioned in this research, especially when a Z specification block and its equivalent NetLogo code are almost of similar length. However, what makes a Z specification a more appropriate means of communication than NetLogo code is that it is based on first-order predicate logic. This makes the Z notation easy to learn, read, understand and remember. On the other hand, in order to understand the logic behind NetLogo source code, the reader is required to learn quite a large amount of syntax as well. Consequently, like with many other programming languages, it is not easy and straightforward to learn NetLogo and more importantly memorise its syntax for future references, modifications and maintenance.

In this paper, we present an approach for systematically translating Z specifications to NetLogo code that was developed while working on the CNAFS case study. We begin in Section 2 with an overview of NetLogo syntax. In Section 3 we provide the translation scheme and illustrate its application on part of the CNAFS case study in Section 4. We then conclude with a discussion of lessons learnt and future directions in Section 5.

2 Overview of NetLogo

In this section we describe the syntax of NetLogo [11] relevant to the case study described in Section 4. A comprehensive documentation of NetLogo can be found in the NetLogo User Manual [12].

2.1 Agents

NetLogo allows developers to define specific *breeds* of agents using the syntax

$\texttt{breed}[\mathit{MyBreeds}\ \mathit{mybreed}]$

where $\mathit{MyBreeds}$ is the plural, or *agentset*, form of the breed name and $\mathit{mybreed}$ is the singular form.

Each type of breed can have its own *properties*. These are a set of attributes that are specific to agents of the breed and, for a given agent, can only be accessed and modified by the agent itself. For example, the following defines agents of type $\mathit{MyBreeds}$ as having properties x, y and z.

$\mathit{MyBreeds}\texttt{-own}[x\ y\ z]$

A number of commands exist for creating and accessing agents. For example,

$$\begin{array}{l} \texttt{create-}\mathit{MyBreeds}\ 2\ [\texttt{set}\ x\ 0 \\ \qquad\qquad\qquad\quad \texttt{set}\ y\ 100] \end{array}$$

creates two agents of type $\mathit{MyBreeds}$ where the brackets enclose a sequence of tasks which are applied to the agents upon creation. In this case NetLogo's `set` keyword is used to set the agents' property x to 0 and property y to 100.

In NetLogo, in order to update the property values of an agent, the `ask` command, which takes an agent or agentset as its input, is used. Thus, the value of z can be set to 10 for all existing agents of type *MyBreeds* as follows.

```
ask MyBreeds [set z 10]
```

2.2 Procedures

Procedures in NetLogo enable developers to modularise their code. A procedure includes a group of statements that aims to perform a particular task on agents, their environment, interface controls, inputs or outputs to the system. NetLogo procedures can be defined either as a *reporter* or as a *command* procedure. A reporter procedure is one that reports a value when it is called somewhere in the code, whereas a command procedure only performs some tasks. Additionally, both reporter and command procedures can take input variables. When a procedure, which takes n inputs, is called elsewhere in the code, the first n words after the procedure name, are considered as its inputs.

Procedures of each type can be defined using the syntax

```
to mycommand [text]                      to-report myreporter
  print text                               report "Hello"
end                                      end
```

where *mycommand* is a command procedure that takes *text* as its input and, using the `print` command, prints it in the NetLogo Command Center which is part of NetLogo's interface. Also, *myreporter* is a reporter procedure that uses the `report` keyword to report a string value ("Hello"). Having defined these two procedures, the following code prints "Hello" to the Command Centre

```
mycommand myreporter
```

where the value reported by *myreporter* is passed to *mycommand* as an input.

2.3 Data structures

NetLogo is an untyped programming language which allows a variable to take different types of values whenever required. In order to define a variable, therefore, it is not required to identify its data type. This section describes the main types of data structures that have been used in the case study.

Globals are those types of variables in the system that can be accessed by all procedures. Such variables can be defined either by using the `globals` keyword as follows

```
globals[myglobal1 myglobal2]
```

or by assigning a name to an interface control such as a slider or switch, which can then be treated as a global variable throughout the code. It should be noted that the defined breeds are also accessible globally.

Locals, on the other hand, are the variables that can only be accessed within the scope of the procedure in which they have been defined. A local can be defined using the `let` keyword and is accessible to following statements within the procedure.

```
to myprocedure
  let mylocal myvalue
    ...      ; other statements
end
```

Strings, numbers, booleans and lists are the main data types that exist in NetLogo. For instance, a string, number and boolean can be defined as follows.

```
let mystring "my string value"
let mynumber 1000
let myboolean? false
```

It is common in the NetLogo user community to add a '?' to the end of a boolean variable name, however it is not compulsory.

Lists allow developers to define more complex data structures. Each element of a list can be a number, string, agent, agentset or a list. A list can be defined as follows.

```
let mylist list 1 2        ; a list with the two elements 1 and 2
```

2.4 Operators and reporters

NetLogo supports the usual range of arithmetic (e.g., $+$,$-$,$*$,$/$), comparison (e.g., $<=$, $>=$, $=$ and $!=$) and logical operators (e.g., `and`, `or` and `not`). NetLogo also has a range of built-in reporters that are explained in the rest of this section.

The `with` reporter can be used to report only those agents from an agentset that satisfy the given conditions as follows

```
ask MyBreeds with [x = 10] [set z 10]
```

where z will be set to 10 only for those agents with $x = 10$. The `with` reporter can be used together with all of the following reporters when required.

The `one-of` reporter can be used to randomly choose a single agent from an agentset. For example, the following equates to an agent with $x = 10$.

```
one-of MyBreeds with [x = 10]
```

The `min-one-of` reporter can be used to randomly choose an agent with the minimum value for a given property. For example, the following equates to an agent with the minimum value of z out of all those agents with $x = 10$.

```
min-one-of MyBreeds with [x = 10] [z]
```

Note that in both the `one-of` and `min-one-of` examples, a reserved value in NetLogo, `nobody` (representing no agent), is reported in the case where no *mybreed* with $x = 10$ is found.

Additionally, whenever it is required to get the value of any agent's properties, the `of` reporter can be used. For example, the x property of an agent can be accessed as follows

[x] `of` *mybreed* 0

where *mybreed* 0 refers to the agent of type *MyBreeds* with `who` number equal to 0. Note that the `who` number is a unique non-negative integer that is automatically assigned to agents when they are created, no matter which agentset they are from.

The `member`? reporter can be used to check that an agent *mb* (defined, for example, as a local variable) is a member of the agentset *MyBreeds* as follows.

`member`? *mb* *MyBreeds*

The `all`? or `any`? reporters, which report `true` or `false`, can be used to check conditions on all or any agents in an agentset. For example,

`set` *myboolean*? (`all`? *MyBreeds* `with` [`color` = `green`] [$x = 0$])

sets *myboolean*? to `true` when all agents of type *MyBreeds* with `color` `green` have their x property equal to 0, or when there is no `green` *mybreed*[1]. Otherwise, *myboolean*? will be set to `false`. Also,

`set` *myboolean*? (`any`? *MyBreeds* `with` [`color` = `green` `and` $x = 0$])

sets *myboolean*? to `true` when at least one agent of type *MyBreeds* with `color` `green` and $x = 0$ exists. Otherwise, *myboolean*? will be set to `false`.

2.5 Branching

The main branching structures in NetLogo, as in most programming languages, are the `if` and `ifelse` commands. The latter can be used to control the flow of the program under two opposite conditions as follows.

```
ifelse mytotal < 1000
[create-MyBreeds 1 [set color green]]
[ask MyBreeds with [color = green] [die]]
```

In this example, if *mytotal* is less than 1000, the commands within the first brackets will be executed and as a result, one agent of type *MyBreeds* will be created and its initial `color` will be set to `green`. However, if *mytotal* is greater than or equal to 1000, then the commands inside the second brackets will be executed and consequently, all the green *MyBreeds* will `die`. The `die` command can be applied on all agents of the system and removes the specified agent from its agentset.

[1] `color` is a property of all agents, and `green` is a constant that may be assigned to `color`.

3 Translating Z to NetLogo

The goal of this section is to describe how a Z specification can be systematically translated into NetLogo code. We adopt the guarded (or blocking) interpretation of Z [4] in which operations can only occur when their pre-state predicates, i.e., their predicates describing the state before the operation, hold. In the traditional (or non-blocking) interpretation of Z, operations can always occur but have an undefined effect when their pre-state predicates do not hold.

It should be noted that not all of the Z notation has been investigated in this work. Rather we have considered a subset of Z that we believe satisfies our requirements of being easy to learn, read, understand and remember while also being adequate for modelling the kinds of systems we are targeting. In particular, all updates of variables are written in the form $x' = e$, where e is an expression, facilitating translation to NetLogo `set` commands. Similarly, all initialisations of variables are written $x = e$. Also, some constructs which are not readily translated are avoided. For example, nested quantifiers are avoided in operation guards. Also, use of *promotion* schemas (used in Z to promote operations on local state spaces to the global system state) is avoided by specifying all operations directly on the global system state.

Additionally, as in other programming languages, there are alternative ways to implement a single task in NetLogo, each of which differs in terms of performance, efficiency, readability and other characteristics. Consequently, the translation examples in this section are not necessarily the best or the most efficient way to implement a Z specification. Instead, they represent how a Z specification could be translated into NetLogo code effortlessly. In this section, we use a car racing game as an example.

3.1 Type definitions

In addition to the predefined types such as $\mathbb{N}$ (natural numbers) and $\mathbb{Z}$ (integers), Z also supports definition of other types [13, 10], such as *free types*. Free types represent the fact that a variable of this type can take a value from the set of distinct specified constants. For instance,

$$LicenceClass ::= Car \mid Lightrigid \mid Mediumrigid \mid Heavyrigid$$

represents a type for specifying different kinds of a driver's licence.

Schemas in Z can also be used as (record) types. This is useful for expressing more details regarding the format of a defined type. For instance, the following schema defines a *Driver* type

```
┌─ Driver ──────────────────────
│ licence : LicenceClass
│ age : ℕ
└───────────────────────────────
```

where *licence* and *age* represent the driver's licence type and age respectively.

Since NetLogo does not support type definition, it is the implementer's responsibility to ensure that the values of variables of such types satisfy the specified constraints throughout the program.

3.2 Global constants

Z supports the definition of global constants which are accessible throughout a specification. They are defined using an axiomatic definition as follows

$$\begin{array}{|l} \mathit{SPEED\_LIMIT} : \mathbb{N} \\ \hline \mathit{SPEED\_LIMIT} = 200 \end{array}$$

where $\mathit{SPEED\_LIMIT}$ represents the highest speed allowed for cars on a road.

In NetLogo, global constants can be defined like global variables using the following syntax.

```
globals[SPEED_LIMIT]
```

The value of $\mathit{SPEED\_LIMIT}$ should then be set in the first procedure that will be run in the NetLogo code (usually called *setup*), so that its value can be used throughout the program. This value should not be changed anywhere else in the code as it is a constant.

```
to setup
    set SPEED_LIMIT 200
    ...    ; other tasks, which should be performed in the setup procedure
end
```

3.3 State and initial state schemas

As mentioned in Section 3.1, schemas can be used as types in Z. State schemas are also used for specifying the main entities of a system. In our car racing game, cars are the main entities (agents) of the system and are specified with the following state schema

$$\begin{array}{|l} \hline \mathit{Car} \\ \hline \mathit{ID} : \mathbb{N} \\ \mathit{fuelAmount} : \mathbb{N} \\ \mathit{speed} : \mathbb{N} \\ \hline \mathit{speed} \leq \mathit{SPEED\_LIMIT} \\ \hline \end{array}$$

where ID, *fuelAmount* and *speed* (in the declaration part of the schema) represent the car's unique ID in the race, amount of fuel and speed respectively. In NetLogo, the main system's entities can be implemented as *breeds* of agents using the following syntax.

```
breed [Cars Car]
Cars-own [ID fuelAmount speed]
```

In Z, the invariant part of the Car state schema ($\mathit{speed} \leq \mathit{SPEED\_LIMIT}$) is implicitly included in all other schemas in which Car is included. However, in NetLogo, such invariants need to be implemented explicitly. For example, whenever the *speed* variable changes, the programmer needs to check its new value to ensure that it satisfies the specified constraint.

State schemas are also used to model the entire system of agents. For example, given the type definition

$$GameStatus ::= Normal \mid Dangerous$$

$CarRacingGame$ is a multi-agent system with a set of $cars$ as the agents of the system and $status$ as the game status.

$$\begin{array}{|l}
\text{\_ } CarRacingGame \text{ \_\_\_\_\_\_\_\_\_\_} \\
cars : \mathbb{P}\, Car \\
status : GameStatus \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\text{\_ } InitCarRacingGame \text{ \_\_\_\_\_\_\_\_\_\_} \\
CarRacingGame \\
\hline
status = Normal \\
\forall\, c : cars \bullet \\
\quad c.fuelAmount = 100 \land c.speed = 0 \\
\hline
\end{array}$$

In NetLogo, the variables of the multi-agent system schema can be defined as globals (as described in Section 2.3).

The $InitCarRacingGame$ specifies that the game $status$ is $Normal$ in the initial state of the system. This can be implemented by setting the value of the global variable $status$ to $Normal$ at the beginning of the program (usually in the $setup$ procedure). The next predicate starts with a universal quantifier ($\forall$), where the $\bullet$ symbol reads *such that* and states that there are some constraints on the quantified variable c. The constraint part of the predicate then specifies that, in the initial state of the system, each member (c) of the $cars$ set has a fuel amount of 100 ($c.fuelAmount = 100$) and a speed of 0 ($c.speed = 0$). These values can be set when the agents are created as described in Section 2.1.

3.4 Operation schemas

In NetLogo, operation schemas of Z can be implemented using procedures. As an example, consider the following operation schemas on the state space of $CarRacingGame$.

Assume that for safety reasons, all moving cars should have a fuel amount higher than 10. If this is the case, the game status would be $Normal$; otherwise, the game status would be $Dangerous$ and one of the unsafe cars is reported. In Z, a variable followed by ! specifies an output of the operation. Also, the Δ symbol represents that one or more variables of the following state schema will be changed as a result of the operation being performed. Note that the post-state variables in Z are displayed using the prime symbol ($'$).

$$\begin{array}{|l}
\text{\_ } GameStatusNormal \text{ \_\_\_\_\_\_\_\_\_\_} \\
\Delta CarRacingGame \\
\hline
\forall\, c : cars \bullet c.speed > 0 \Rightarrow \\
\quad c.fuelAmount > 10 \\
status' = Normal \land cars' = cars \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\text{\_ } GameStatusDangerous \text{ \_\_\_\_\_\_\_\_\_\_} \\
\Delta CarRacingGame \\
unsafe! : Car \\
\hline
\exists\, c : cars \bullet c.speed > 0 \land \\
\quad c.fuelAmount \leq 10 \land unsafe! = c \\
status' = Dangerous \\
\hline
\end{array}$$

The *GameStatusNormal* and *GameStatusDangerous* operation schemas can be implemented in NetLogo as follows. Note that in translating an operation no action is required if a variable remains unchanged (e.g., as in the predicate $cars' = cars$).

```
to game-status-normal
    if all? Cars with [speed > 0][fuelAmount > 10]
    [set status "Normal"]
end
to-report game-status-dangerous
    ifelse any? Cars with [speed > 0 and fuelAmount <= 10]
    [report one-of Cars with [speed > 0 and fuelAmount <= 10]
     set status "Dangerous"]
    [report nobody]
end
```

As can be seen, we use nearly direct translation from the quantified expressions of the operation schemas in Z to the NetLogo procedures' statements. These expressions are guards of the operations and hence checked using an `if` or `ifelse` statement. The `with` reporter can be used in the `all`? statement to introduce constraints on the quantified variable. Such constraints would appear in Z as proposition $P(x)$ in predicates of the form $\forall x : X \mid P(x) \bullet Q(x)$ or $\forall x : X \bullet P(x) \Rightarrow Q(x)$. The translation of $Q(x)$ comes within the last brackets in the `all`? statement. Similar constraints $P(x)$ in Z predicates of the form $\exists x : X \mid P(x) \bullet Q(x)$ appear in the single set of brackets after the `with`, combined with the translation of $Q(x)$ using `and`. To access an existentially quantified variable, such as c in *GameStatusDangerous*, we utilise the `one-of` reporter. Note that if the existentially quantified variable is required to have the minimum value for a given property (as in the case study of Section 4) we use the `min-one-of` reporter instead.

Whenever a Z operation has an output, it needs to be translated as a reporter procedure in NetLogo. Hence, the *game-status-dangerous* procedure is defined as a reporter. The output is `nobody` in the case where the Z operation's guard is false.

In Z such outputs can be used as inputs to other schemas using the piping operator ($\gg$) [10]. For example, *RefuelUnsafe* specifies an operation in which an unsafe car is refuelled. In this operation the output *unsafe*! of *GameStatusDangerous* is equated with the input *unsafe*? of *Refuel*. In Z, a variable followed by ? denotes an input to the operation.

Refuel

$\Delta CarRacingGame$
$unsafe? : Car$

$unsafe? \in cars$
$\exists\, uc : Car \bullet uc.ID = unsafe?.ID \wedge uc.fuelAmount = 100 \wedge uc.speed = 0$
$\quad \wedge\ cars' = cars \setminus \{unsafe?\} \cup \{uc\}$

$RefuelUnsafe \mathrel{\widehat=} GameStatusDangerous \gg Refuel$

The $\exists$ quantifier in *Refuel* is used to define a new car uc, which has the same ID as the *unsafe*?, fuel amount of 100 and speed of 0. The last part of the predicate specifies that the new car uc is replaced with the unsafe car *unsafe*? in the *cars* set. The union symbol ($\cup$) can be translated into NetLogo code by creating a new agent. This agent will automatically be added to the agentset. Also, the set difference symbol ($\setminus$) is translated by using the `die` command which removes the old agent from the agentset. Hence, the above operations can be translated as

```
to refuel [unsafe]
    if (member? unsafe Cars)
    [create-Cars 1 [set ID ([ID] of unsafe)
                   set fuelAmount 100
                   set speed 0]
     ask Cars with [self = unsafe][die]]
end
to refuel-unsafe
    if game-status-dangerous != nobody
    [refuel game-status-dangerous]
end
```

where `self` is a reporter used to refer to the current agent at each iteration of the *ask* command. Note that equality between two agents ($=$ operator) is checked according to their `who` numbers. Additionally, in order to access the state variables of a variable that is of type schema in Z a dot (.) is used. This dot can be translated using the `of` reporter in NetLogo, e.g., $unsafe?.ID$ in Z is translated into $[ID]\ of\ unsafe$.

4 Case study

In this section, we illustrate the translation scheme on a small part of the CNAFS case study: a model of movement of *chyme*, i.e., food undergoing digestion, through the small intestine of a pig. In their experiments, the researchers consider the small intestine as comprising 6 different intestine segments (SI1 – SI6). One of the main reasons for this segmentation is that the movement rate of chyme varies in each of these segments. To allow results of the simulation to be verified against experimental data, most parts of the specification are based on CNAFS researchers' hypotheses and their methods of running their experiments. Additional biological details of small intestine functionality are derived from Guyton and Hall [5]. Using the built-in NetLogo visualisation facilities, the outcome of this simulation provides the biologists with a visualisation of the system at each time step and some statistical results, such as total amount of chyme content and marker content in each intestine segment at each time step.

4.1 State definitions

All non-schema types used in this section are defined as appropriate global types in Z. The agents of the system are intestine segments and packets of chyme. The idea of considering chyme as a collection of discrete packets is derived from the functionality of the *pyloric valve* which controls chyme entry to the small intestine [5].

An intestine segment is specified in terms of its length, the total amount of chyme content that exists in the segment, and the movement rate of chyme packets in the segment. Also, each segment can only take up to a certain amount of chyme because of physical limits on its expansion. This value is represented by $contentThreshold$. When the total amount of chyme content in a segment reaches this threshold, the variable $entryBlocked$ of the segment will be set to Yes to specify that the segment cannot take any more packets. The value of $entryBlocked$ will be changed back to No whenever the total amount of chyme content is decreased to a value less than $contentThreshold$.

$$\begin{array}{|l}
\multicolumn{1}{l}{IntestineSegment} \\ \hline
length : NonNegativeReal \\
totalExistingChymeContent : NonNegativeReal \\
chymePassageRate : NonNegativeReal \\
contentThreshold : NonNegativeReal \\
entryBlocked : YesOrNo \\ \hline
totalExistingChymeContent \geq contentThreshold \Leftrightarrow entryBlocked = Yes \\ \hline
\end{array}$$

A schema $Position$ represents a chyme packet's current position in the small intestine. In the $Position$ schema, $segNum$ represents the ID of the segment that the packet is currently in and $posInSeg$ specifies the packet's distance from the beginning of the segment. Each chyme packet contains specific amounts of nutrients, markers and water. Markers are consumable, but non-absorbable materials used in experiments for different purposes such as calculation of passage rate in the gastrointestinal tract [7]. All these contents together have a total mass that is represented by the variable $totalContent$.

$$\begin{array}{|l}
\multicolumn{1}{l}{Position} \\ \hline
segNum : SegmentID \\
posInSeg : NonNegativeReal \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{ChymePacket} \\ \hline
Position \\
nutrients : \mathbb{P}\, Nutrient \\
markers : \mathbb{P}\, Marker \\
waterAmount : NonNegativeReal \\
totalContent : NonNegativeReal \\ \hline
\end{array}$$

The (multi-agent) system is a small intestine comprising a sequence of intestine segments and set of chyme packets that have entered, but not left the small intestine. The variables $totalLength$, $chymeEntryRate$ and $emptyingBlocked$ represent the small intestine length, the rate at which the chyme packets enter the small intestine and whether the packets can leave the small intestine or not, respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SmallIntestine}} \\
\hline
\textit{segments} : \textit{SegmentID} \rightarrow \textit{IntestineSegment} \\
\textit{chymePackets} : \mathbb{F}\ \textit{ChymePacket} \\
\textit{totalLength} : \textit{NonNegativeReal} \\
\textit{chymeEntryRate} : \textit{NonNegativeReal} \\
\textit{emptyingBlocked} : \textit{YesOrNo} \\
\hline
\forall\, c1, c2 : \textit{chymePackets} \bullet \\
\qquad c1.\textit{segNum} = c2.\textit{segNum} \land c1.\textit{posInSeg} = c2.\textit{posInSeg} \Leftrightarrow c1 = c2 \\
(\textit{segments}\ 1).\textit{length} = (\textit{segments}\ 6).\textit{length} = 1 \\
\forall\, \textit{segID} : \textit{SegmentID} \bullet \textit{segID} \neq 1 \land \textit{segID} \neq 6 \Rightarrow \\
\qquad (\textit{segments}\ \textit{segID}).\textit{length} = (\textit{totalLength} - 2) \operatorname{div} 4 \\
\forall\, c : \textit{chymePackets};\ \textit{segID} : \textit{SegmentID} \bullet \\
\qquad c.\textit{segNum} = \textit{segID} \Rightarrow c.\textit{posInSeg} \leq (\textit{segments}\ \textit{segID}).\textit{length} \\
\hline
\end{array}$$

The predicate of *SmallIntestine* states that no chyme packets have the same position. Additionally, according to the experiments at CNAFS, both the first and the last segments (SI1 and SI6) of the small intestine are considered to be 1 metre long and the other four segments are each one quarter of the remaining length of the small intestine. Finally, the position of each chyme packet in each segment must be less than or equal to the segment length.

When translating a schema such as *ChymePacket* that includes another schema, we include the variables of the included schema as properties of the NetLogo breed. When translating collections of agents such as *segments* which are specified in terms of a function, we include the domain value associated with an agent, as a property of the NetLogo breed. Effectively, we are using the Z interpretation of the function as a set of ordered pairs of domain and range values [10]. Hence, the NetLogo translation of the above is as follows. As mentioned in Section 3.3, state invariants need to be implemented explicitly in operations.

$\textit{breed}\ [\textit{IntestineSegments IntestineSegment}]$
$\textit{breed}\ [\textit{ChymePackets ChymePacket}]$
$\textit{IntestineSegments-own}\ [\textit{segmentID length chymePassageRate}\ \ldots]$
$\textit{ChymePackets-own}\ [\textit{segNum posInSeg nutrients markers}\ \ldots]$
$\textit{globals}\ [\textit{totalLength chymeEntryRate emptyingBlocked}\ \ldots]$

4.2 Operations

This section describes the case in which a chyme packet wants to move through one intestine segment, but will be blocked by another packet. One of the assumptions made in the specification is that chyme packets move through and leave the small intestine in the same order as they arrive. Therefore, packets cannot pass each other and sometimes packets may be blocked.

An operation *MovingBlocked* in the Z specification specifies the movement of a packet *pkt*? being blocked by another packet *blocking*! in the same segment. The function *Min* is a predefined global constant in the specification which returns the minimum of a set of real numbers (defined similarly to Z's *min* function for integers [10]).

```
$MovingBlocked$
$\Xi SmallIntestine$
$pkt? : ChymePacket$
$blocking! : ChymePacket$
----------------------------------------------------------------
$pkt? \in chymePackets$
$pkt?.posInSeg +$
      $(segments\ pkt?.segNum).chymePassageRate * TIMESTEP \leq$
            $(segments\ pkt?.segNum).length$
$\exists\, c : chymePackets \bullet$
      $c.segNum = pkt?.segNum \land c.posInSeg > pkt?.posInSeg \land$
      $c.posInSeg \leq pkt?.posInSeg +$
            $(segments\ pkt?.segNum).chymePassageRate * TIMESTEP \land$
      $c.posInSeg = Min(\{ch : chymePackets \mid ch.segNum = pkt?.segNum \land$
                              $ch.posInSeg > pkt?.posInSeg \bullet ch.posInSeg\}) \land$
      $blocking! = c$
```

The first two predicates state that $pkt?$ is a chyme packet in a segment of the small intestines which, if unblocked, would not leave that segment in the next time step ($TIMESTEP$ is a global constant representing the time step in our NetLogo simulation). The final predicate states there exists another packet c which will block $pkt?$'s movement and assigns that packet to the output variable $blocking!$. Following the translation scheme in Section 3, the operation is translated as follows. Note that in order to access agents which are specified in the range of a function, such as $segments$, the `one-of` and `with` reporters are used, where the desired domain value comes inside the brackets after `with`.

```
to-report MovingBlocked [pkt]
    ifelse (member? pkt ChymePackets) and
        ([posInSeg] of pkt +
            ([chymePassageRate] of one-of IntestineSegments with
                [segmentID = [segNum] of pkt] * TIMESTEP) <=
                    [length] of one-of IntestineSegments with
                        [segmentID = [segNum] of pkt]) and
        (any? ChymePackets with [(segNum = [segNum] of pkt) and
            (posInSeg > [posInSeg] of pkt) and
            (posInSeg <= [posInSeg] of pkt +
                ([chymePassageRate] of one-of IntestineSegments with
                    [segmentID = [segNum] of pkt] * TIMESTEP))])
        [report min-one-of ChymePackets with
                [segNum = [segNum] of pkt and
                posInSeg > [posInSeg] of pkt] [posInSeg]]
        [report nobody]
end
```

The operation $MoveUntilBlocked$ specifies that a chyme packet $pkt?$ moves to right behind another packet $blocking?$ which is blocking it.

$$
\begin{array}{|l}
\hline
\textit{MoveUntilBlocked} \\
\Delta \textit{SmallIntestine} \\
\textit{pkt}? : \textit{ChymePacket} \\
\textit{blocking}? : \textit{ChymePacket} \\
\hline
\textit{pkt}? \in \textit{chymePackets} \land \textit{blocking}? \in \textit{chymePackets} \\
\exists\, \textit{updPkt} : \textit{ChymePacket} \bullet \textit{updPkt.segNum} = \textit{pkt}?.\textit{segNum} \land \\
\quad ((\textit{blocking}?.\textit{posInSeg} - \textit{PKTSIZE} > \textit{pkt}?.\textit{posInSeg} \Rightarrow \\
\qquad \textit{updPkt.posInSeg} = \textit{blocking}?.\textit{posInSeg} - \textit{PKTSIZE}) \\
\quad \lor (\textit{blocking}?.\textit{posInSeg} - \textit{PKTSIZE} \leq \textit{pkt}?.\textit{posInSeg} \Rightarrow \\
\qquad \textit{updPkt.posInSeg} = \textit{pkt}?.\textit{posInSeg})) \\
\quad \textit{updPkt.nutrients} = \textit{pkt}?.\textit{nutrients} \land \textit{updPkt.markers} = \textit{pkt}?.\textit{markers} \land \\
\quad \textit{updPkt.waterAmount} = \textit{pkt}?.\textit{waterAmount} \land \\
\quad \textit{updPkt.totalContent} = \textit{pkt}?.\textit{totalContent} \land \\
\quad \textit{chymePackets}' = \textit{chymePackets} \setminus \{\textit{pkt}?\} \cup \{\textit{updPkt}\} \\
\textit{totalLength}' = \textit{totalLength} \land \textit{emptyingBlocked}' = \textit{emptyingBlocked} \\
\textit{segments}' = \textit{segments} \land \textit{chymeEntryRate}' = \textit{chymeEntryRate} \\
\hline
\end{array}
$$

The first predicate of this schema states that *pkt*? and *blocking*? are chyme packets within the small intestine. The second predicate replaces *pkt*? with a new chyme packet *updPkt* which is in the position the blocked packet would move to, and is otherwise identical to *pkt*? (*PKTSIZE* is a global constant representing the size of chyme packets in our NetLogo simulation). The remaining predicates indicate that the small intestine is otherwise unchanged.

MoveUntilBlocked is combined with the operation schema *MovingBlocked*, which provides the input *blocking*?, as follows.

$$\textit{PacketMoveInSegmentBlocked} \mathrel{\widehat{=}} \textit{MovingBlocked} \gg \textit{MoveUntilBlocked}$$

This part of the specification is translated into the following NetLogo code.

```
to MoveUntilBlocked [pkt blocking]
    if(member? pkt ChymePackets) and (member? blocking ChymePackets)
    [create-ChymePackets 1 [
        set segNum ([segNum] of pkt)
        ifelse ([posInSeg] of blocking − PKTSIZE) > ([posInSeg] of pkt)
            [set posInSeg ([posInSeg] of blocking − PKTSIZE)]
            [set posInSeg ([posInSeg] of pkt)]
        set nutrients ([nutrients] of pkt)
        set markers ([markers] of pkt)
        set waterAmount ([waterAmount] of pkt)
        set totalContent ([totalContent] of pkt)]
    ask ChymePackets with [self = pkt] [die]]
end
to PacketMoveInSegmentBlocked [pkt]
    if MovingBlocked pkt != nobody
    [MoveUntilBlocked pkt (MovingBlocked pkt)]
end
```

5 Conclusion

This research combined the use of the Z formal notation with computational modelling in the NetLogo simulation language. This reduced a large amount of effort required for the developer of the simulation to firstly understand the system requirements and functionality clearly, and to secondly efficiently derive code directly from the specification of these requirements. The approach was trialled on a real project studying digestion in pigs' intestines. During simulations, the emergent property of total contents in different segments increased along the intestine in a manner qualitatively in agreement with the patterns seen in experimental data. Additionally, modifications to the model were readily integrated into the Z specification and, via translation, into the NetLogo simulation. Overall, the application of the approach was successful in the sense that it made the development process more convenient for all the people involved. This warrants its ongoing use as well as use in similar projects in the future.

A major lesson learnt is that the usability and effectiveness of formal methods is influenced by human-factors such as the background of the people involved in the development process. Consequently, one important step before applying formal methods is to choose a suitable formal modelling language that makes the software development process more efficient and convenient for all the people involved.

Acknowledgements This project was jointly supported by the Queensland Alliance for Agriculture and Food Innovation (QAAFI) and Australian Research Council (ARC) Discovery Grant DP110101211.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language user guide. *Reading, PA: Addison-Wesley*, 1999.
3. J.P. Bowen. *Formal specification and documentation using Z: A case study approach*. International Thomson Computer Press London, 1996.
4. J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, 2nd edition, 2014.
5. A.C. Guyton and J.E. Hall. *Guyton and Hall Textbook of Medical Physiology*. Saunders Elsevier, 12th edition, 2011.
6. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
7. F.N. Owens and C.F. Hanson. External and internal markers for appraising site and extent of digestion in ruminants. *Journal of Dairy Science*, 75(9):2605–2617, 1992.
8. V.K. Singh, D. Gautam, R.R. Singh, and A.K. Gupta. Agent-based computational modeling of emergent collective intelligence. In *Computational Collective Intelligence*. Springer, 2009.
9. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
10. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
11. S. Tisue and U. Wilensky. Netlogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, pages 16–21, 2004.
12. U. Wilensky. *NetLogo User Manual*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 5.0.5 edition, 2013.
13. J.C.P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1994.