University of Glasgow | School of Computing Science

Analysis of a C++ Limbic System Simulator With The Spin Model Checker

Alex Trew 2090328

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 2018

Contents

Chapter 1

Introduction

1.1 Objectives

In this report, our goal is to analyse an agent-based learning system implemented in the C++ programming language. Specifically, a simulation of a rat's limbic system. We will investigate the process of *model checking* and *abstraction* with regard to complex code written in the C++ programming language. We will investigate the efficacy of, using various methods, abstracting this program into a *model* , where the functionality of the model will faithfully express that of the original system while being computationally simple enough to be verified using a finite state model checker. We will attempt to verify various properties of the system, namely, whether or not the rat can always learn to find its reward rather than simply randomly find it by chance, and whether or not it can unlearn the location of a reward. To achieve the highest quality model of the limbic system, we first need to investigate the best way translate C++ into a model.

1.2 Motivation

Formal verification is an extremely useful tool for increasing the quality of software because it allows developers to both discover obscure errors, and gain an understanding of the behaviour of a system to a great extent. Formally verifying the properties of a given system allows us to make definitive statements as to whether or not the specification of that system has been met.[22] Despite the vast improvements of verification algorithms and the fairly small learning curve to certain systems of model checking, such as the Spin model checker, many developers shy away from their perceived complexity [22]. There are a few different approaches of doing this we will address throughout this report. These will mainly be split into two categories: automatic and manual translation. Fully understanding the behaviour of the limbic system simulator by formally verifying that it operates as intended, will mean that it will be significantly more valuable for future experiments.

In the past, analysis of agent-based learning systems has been done through running sets of simulations and observing behaviour [19][13], however in many cases, approximations or speculation as to the behaviour of a system under certain conditions can be an unreliable and unsatisfactory method of testing. This is particularly true with regard to expensive or safety critical systems.[19] A formal method of verification such as *finite state model checking* for systems such as these is ideal, for it allows us to verify that properties hold under every possible state of execution, giving one a far greater degree of certainty that the program will operate correctly once deployed. As C++ is often used in the development of such systems [21], such as, for example, the flight codes for the crew exploration vehicle Orion [25], being able to verify programs implemented in the C++ language is a very important area of research to pursue.

Chapter 2

Background

2.1 The Limbic System Simulator

The limbic system simulator program (which we will refer to as the *lss*) is a simple simulator for reversal learning and reward-based learning in animals. Simply put, it consists of an agent exploring an environment. It uses input signals to generate an output signal, which represents an association between a particular landmark and a reward, and directs the agent to that landmark.

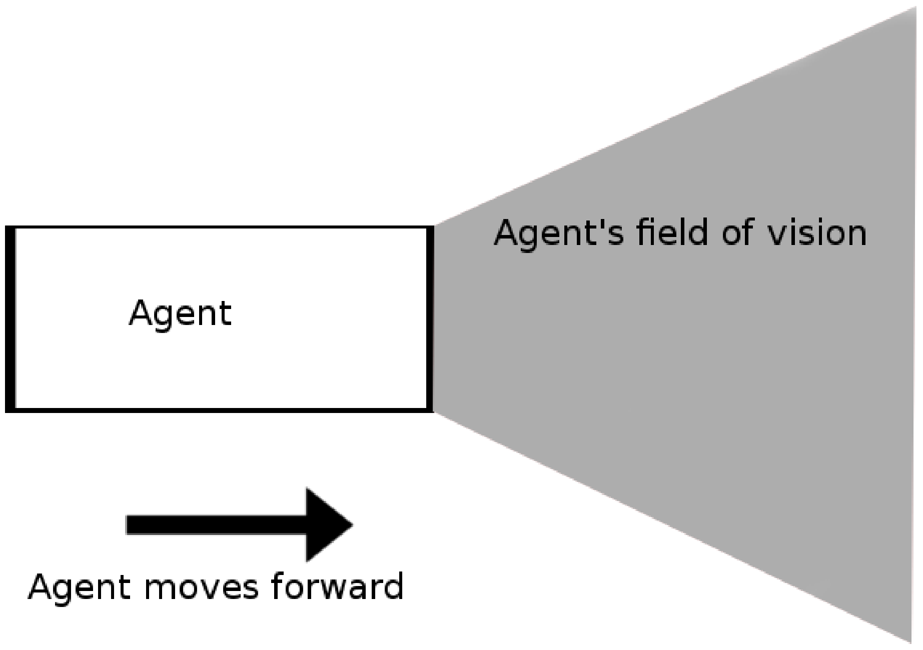


Figure. 1. Simple diagram depicting the agent and its field of view.

The agent can detect objects that fall within its field of view. From these objects, it generates several input signals: a reward signal, visual input signal two landmarks, denoted as the light and dark green landmarks, a signal indicating that the agent is within one of the landmark areas, and a visual input when the agent sees the reward.

The agent randomly explores the area and takes the visual input signal when it sees either landmark, and a reward signal when it successfully consumes the reward item. The reward value is used to build an association between the landmark containing the reward item and the reward. It uses a combination of the visual input signals and the reward signal to gradually generate an output signal which influences the agent's motor signals which, when higher than a certain value, directs the agent to that landmark. As the agent successfully repeats this action, the output signal directing it to the landmark grows stronger, hence the association grows stronger. However, after a given number of iterations of the reward item switches places with a fake reward item. When the agent moves to the landmark now containing the fake reward, it neither gets a reward nor generates the reward signal. As a result, the output signal generated will gradually fade until the motor signals are low enough for the agent to start randomly exploring again. At this point, the agent will randomly explore until it is able to build an association between the landmark containing the reward and the reward. A technical description of the *lss* will be given in chapter 4.

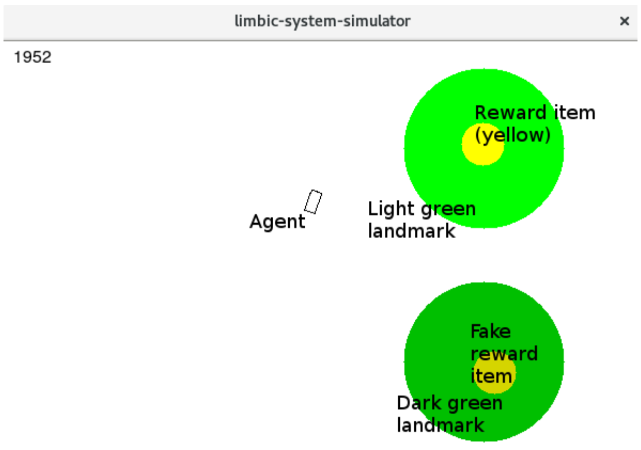


Figure. 2 . The agent navigating the explorable area.

2.2 Model Checking Overview

Model checking is an automatic technique for verifying finite-state systems. Model checking involves creating an abstraction of a system known as a model, the properties of which can then be verified by applying linear time temporal logic formulae. [14]

2.3 Linear Time Temporal logic and Büchi Automata

We will be expressing our desired properties using a temporal logic, specifically, *linear time temporal logic* (LTL). To understand LTL we must first define CTL*, or Computational tree logic. CTL* is a set of state formulas where the CTL* state and path formulas are defined inductively to follow. The quantifier A denotes "For all paths" and E "for some paths". The operators X, U, $<>$, and $[]$ represent *nexttime, strong until, eventually, and always* respectively where $<> \phi = trueU\phi$ and $[]\phi = \neg <> \neg\phi$.

LTL is a subset of CTL* where the set of CTL* formulas is restricted to those of the form $A\phi$ where ϕ does not contain A or E. We normally omit the A operator and interpret ϕ as "for all paths phi". This being the case, LTL is used for representing all possible states of a given model.

The Model checking problem for LTL is as follows: "given M and ϕ, does there exist a path that does not satisfy ϕ"

We would say that M satisfies ϕ (i.e $M\phi$), if all of the initial states of M satisfy ϕ. [20] [19]

As an LTL formula represents an infinite run of a *finite-state* automaton, we say that and LTL formula can be expressed as a Büchi automaton, which satisfies Büchi acceptance (or is an accepting $\omega - run$) if some state in F if and only if is visited infinitely often in the run.

An example of an LTL statement expressed in Promela, the language interpreted by the Spin model checker, is as follows:

```
#define p (program_has_reached_x==1) //define p as being true,
//when program has reached x

//rest of program

ltl p1{[]<>p} //ltl claim be checked by the model checker.
```

Below is the LTL claim expressed in English:

*For every path of execution, from **any** state x will **eventually** be true.*

2.4 The Spin Model Checker

The Spin Model Checker is an open source generic verification system targeting the efficient verification of multi-threaded software developed at Bell Labs in the Unix group of the Computing Sciences Research Center. Models are specified using a high-level language called Promela (Process Meta Language). Spin works **on-the-fly**, so it doesn't need to construct an entire global state graph or Kripke structure prior to doing a verification run. This means that it is possible to verify very large system models[9].

Spin is an excellent tool for verification. It has become widely used in industries that build critical systems[11]. Spin was used for the verification of BOS project: an automated moveable storm barrier built in order to protect the low, western part of the Netherlands from storm floods[18]. It was also used by the NASA Engineering and Safety Center to test for design and implementation vulnerabilities in the Toyota Electronic throttle System.[10]

The structure of the Spin model checker is illustrated below in Figure. 3.

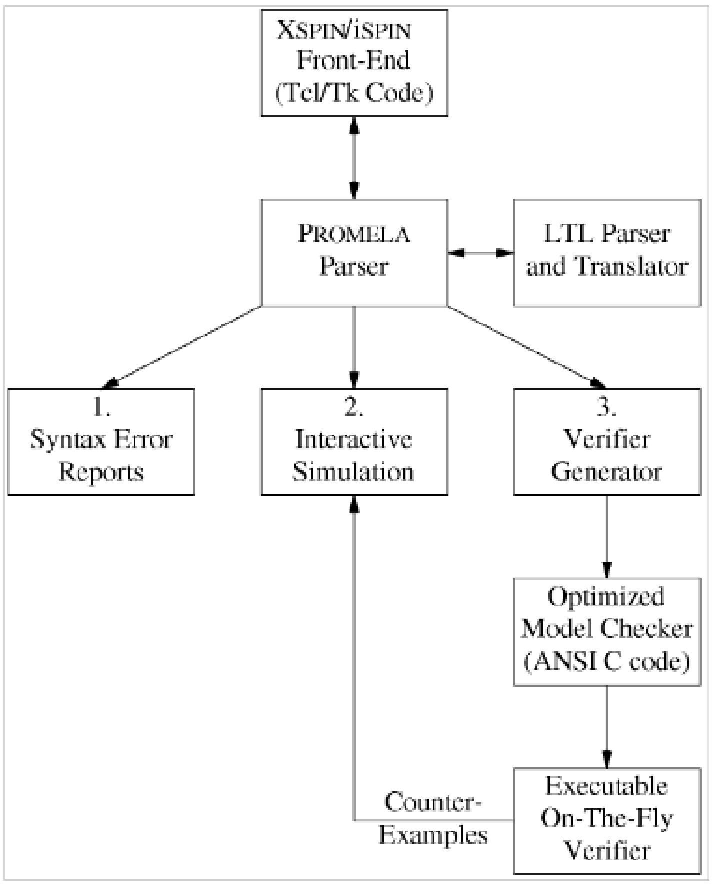

XSPIN/iSPIN Front-End (Tcl/Tk Code)
PROMELA Parser
LTL Parser and Translator
1. Syntax Error Reports
2. Interactive Simulation
3. Verifier Generator
Optimized Model Checker (ANSI C code)
Executable On-The-Fly Verifier
Counter-Examples

Figure. 3. The structure of SPIN simulation and verification.[17]

A user will start with a Promela specification, typically using a graphical interface such as XSpin or iSpin. iSpin has a syntax error checking feature to give the user the necessary information to resolve such errors. Interactive or randomised simulation of the model can be used to give the user a basic idea of how the model behaves. Once the user has a fair amount of confidence that their model behaves as intended, Spin can be used to generate an on-the-fly verification program *pan*. The verifier is then compiled and run. A verification can check for invalid end states, or unreachable code. LTL properties can be included for correctness claims to checked. If any invalid end states or exceptions to the correctness claims are found in the model, an error trail can be replayed as a guided simulation and inspected in detail to understand and eliminate their cause.[17]

In order to improve efficiency, Spin uses a technique known as *partial order reduction* to reduce the size of the state space to be searched by the model checking algorithm. This is because LTL formulae are insensitive to the order in which independently executed transitions are executed, which results in the same state if executed in a different order.[17]

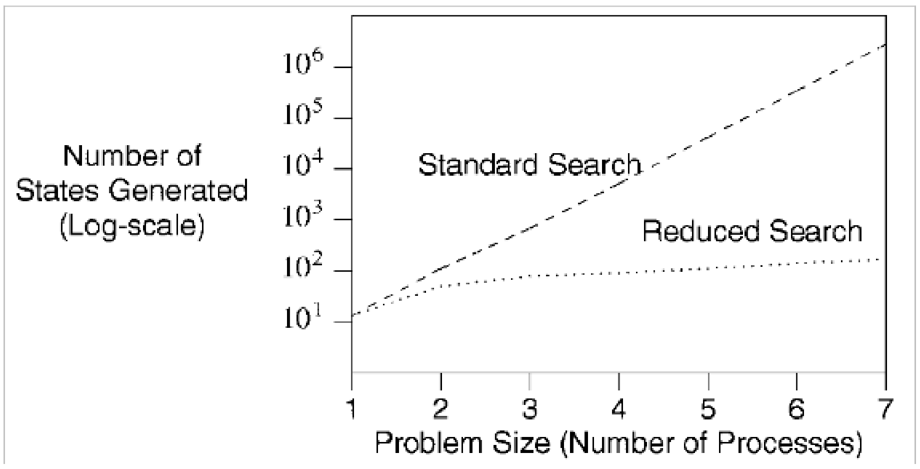


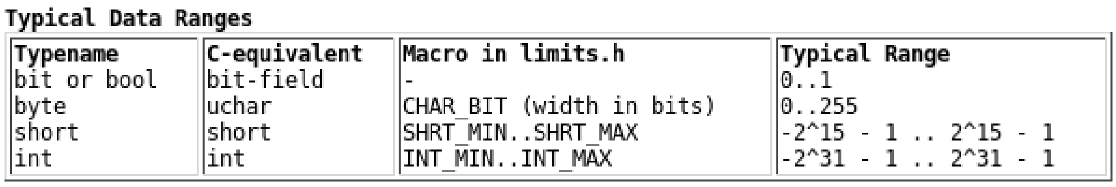
Figure. 4. Effect of partial order reduction on number of reachable states to be reached in a model of a leader election algorithm[17]

2.5 Promela

In this section we discuss the basic semantics of Promela and some relevant features of the language. Promela is a modelling language based on Dijkstra's guarded command language, which uses a C like syntax and allows a user to define processes which can be run concurrently. The processes can communicate using shared global variables and message passing via channels.

Promela uses seven predefined integer types: *bit, bool, byte, pid, short, int and unsigned.* [7] *pid* is used to identify a process that has been defined.

When incorporating an integer into a model, it is often a good idea to use the type with the smallest possible range that is suitable. The reason for this is that using a integer type with a smaller range uses a reduced memory space. For example, storing an *int* uses 4 bytes of memory; double the memory of a *short*, which only uses 2 bytes. Verifying complex models can be inhibitively expensive in terms of memory, so it is good practice to help minimise memory usage by this method.[7]

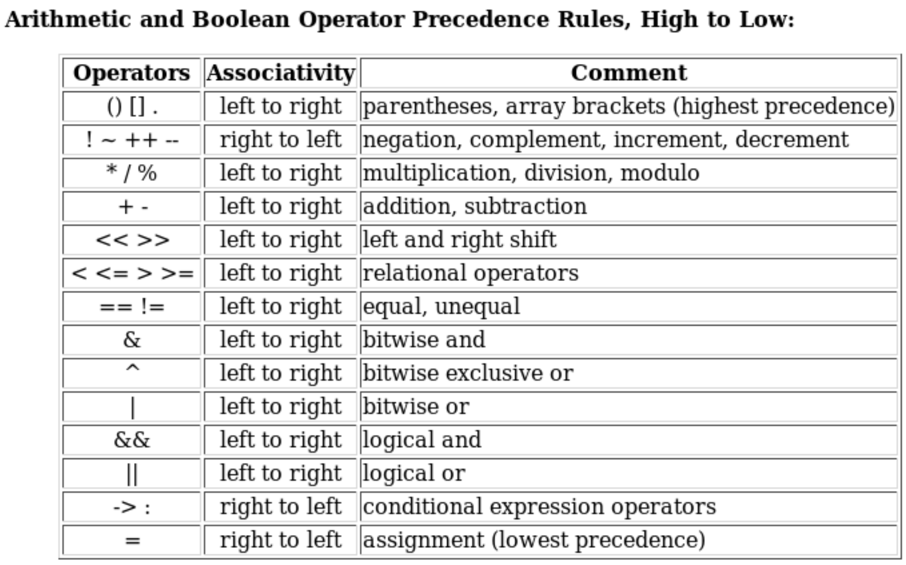


Typical Data Ranges

Typename	C-equivalent	Macro in limits.h	Typical Range
bit or bool	bit-field	-	0..1
byte	uchar	CHAR_BIT (width in bits)	0..255
short	short	SHRT_MIN..SHRT_MAX	-2^15 - 1 .. 2^15 - 1
int	int	INT_MIN..INT_MAX	-2^31 - 1 .. 2^31 - 1

Figure. 5. Ranges of integer types in Promela[7]

Promela uses all of the standard operators found in C like languages


Arithmetic and Boolean Operator Precedence Rules, High to Low:

Operators	Associativity	Comment
() [] .	left to right	parentheses, array brackets (highest precedence)
! ~ ++ --	right to left	negation, complement, increment, decrement
* / %	left to right	multiplication, division, modulo
+ -	left to right	addition, subtraction
<< >>	left to right	left and right shift
< <= > >=	left to right	relational operators
== !=	left to right	equal, unequal
&	left to right	bitwise and
^	left to right	bitwise exclusive or
|	left to right	bitwise or
&&	left to right	logical and
||	left to right	logical or
-> :	right to left	conditional expression operators
=	right to left	assignment (lowest precedence)

Figure. 6. Operators in Promela [3]

Process behaviour is specified via *proctypes*, which are defined using a similar syntax to C functions.[8]

Below is an example of a proctype declaration, which takes the parameter a:

```
proctype A(int a)
{
    //process behaviour
    //...
}
```

[8]

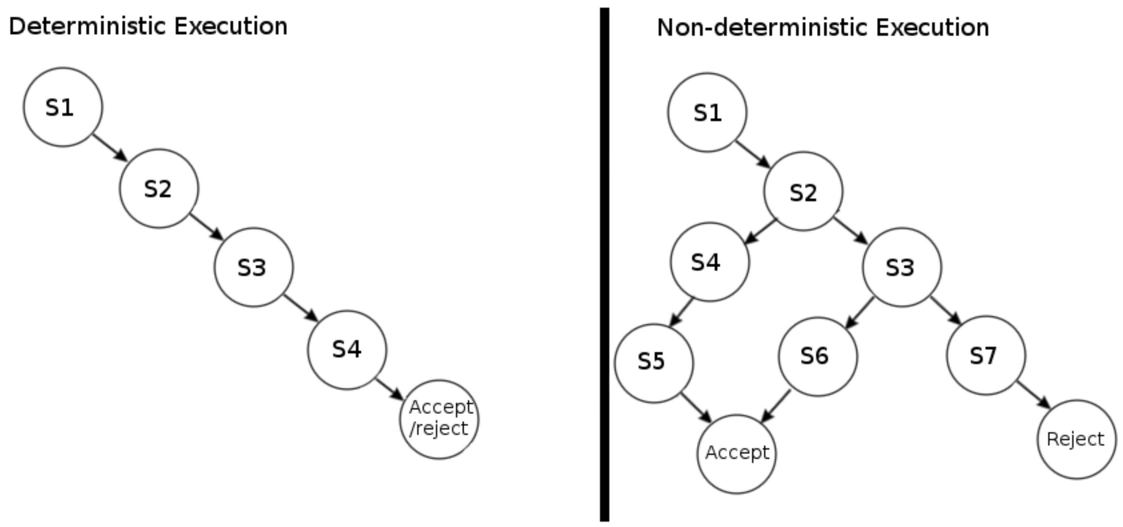
If a proctype is prefixed with the *active* keyword, it will be run at the initial system state, otherwise, it must be run from the *init* process.[8]

The *init* process is a special process which is always active in the initial system state. All other processes can be created from here via the *run()* operator. This operator can be used to dynamically create new processes from any process.[6]

We make use of the *if* and *do* constructs, which are respecively a selection construct, and a repetition construct. Unintuitively,the selection construct does not function the same way as an if statement in C. It is used to define the structure of underlying automation Any number of sequences can be placed within the construct. sequences following a *::* operator will be executed non deterministically. The repetition construct functions similarly to the selection construct; the difference being that it will loop over its contents until the loop is broken. [spin manual]

The first statement in each of these sequences is known as its *guard*. The remainder of that sequence will only be executed if the guard is executable. Using an expression as the first statement in a sequence behaves similarly to a C if statement.[5]

As mentioned above, different paths in Promela can be executed *non-deterministically*. Non-deterministic algorithms make an arbitrary selection from the set of possible values each time a multiple-valued function is encountered.[27]



Non-determinism is an extremely useful tool for verification, as during the verification by Spin, we check every single possible path of execution. This is an effective way of explicitly expressing different paths in the model.

Promela allows a user to define aliases for constant values through the use of *mtypes*. mtypes can represent any data type, and are useful for making Promela code more readable. It functions in the same way as making a series of #define statements in C.

The example given in the Spin user manual is as follows:

The declaration

```
mtype = {ack, nak, err, next, accept}
```

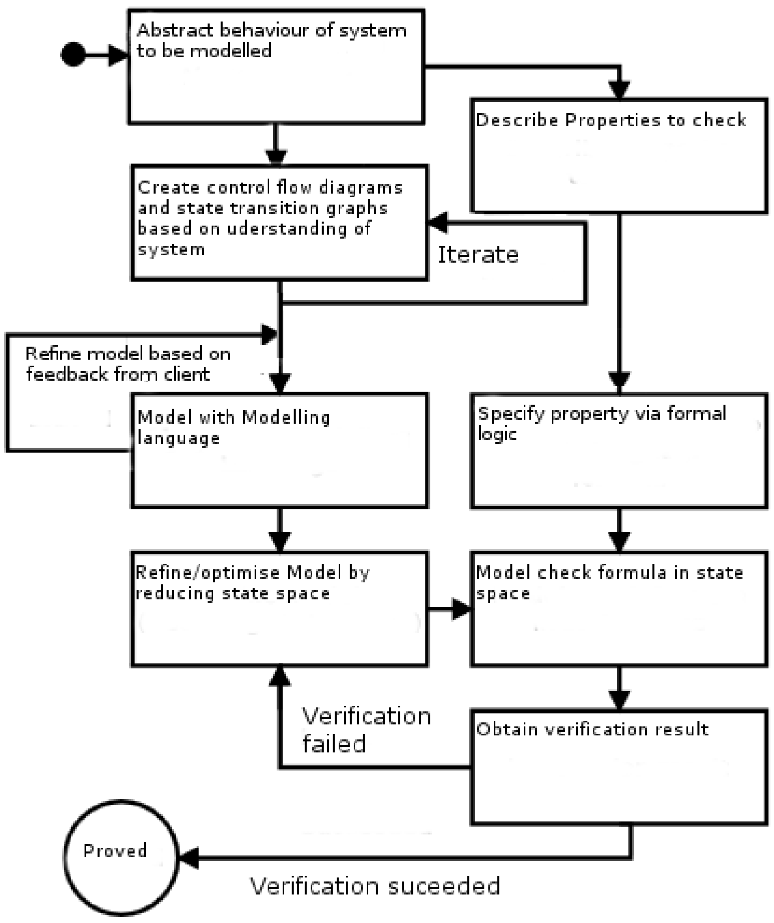
is functionally equivalent to

```
#define    ack     5
#define    nak     4
#define    err     3
#define    next    2
#define    accept  1
```

These can be referred to elsewhere in the code by their alias, similarly to an *enum* in C.

2.6 Creating an Abstraction of a System

The process of model checking starts with some analysis of the software system we wish to model. Once a suitable degree of understanding of the system is achieved, it is translated into a model. This program will be written to represent particular properties of the system whose behaviour we are interested in. Properties of the system are be omitted from the program if they are not relevant or of interest with respect to the specification[19], with the goal of reducing the size of the state transition graph. This program can be considered a simplified abstraction of the original system, and can be represented as a set of states and transitions. This technique is known as *cone of influence reduction*.[15] By using different combinations of these states, we can understand how the system will behave under those conditions. A graph detailing this process is shown below in figure 3:



Abstract behaviour of system to be modelled
Describe Properties to check
Create control flow diagrams and state transition graphs based on uderstanding of system
Iterate
Refine model based on feedback from client
Model with Modelling language
Specify property via formal logic
Refine/optimise Model by reducing state space
Model check formula in state space
Verification failed
Obtain verification result
Proved
Verification suceeded

Figure. 7. Graph showing the process we used to analyse the lss and implement the model.

Chapter 3

Some Methods for the Translation of C++ to Promela

3.1 Model Checking C++

While C code is relatively simple to model (especially as Promela allows C code to be directly embedded in a model) and its verification is well documented, C++ is a very complex language in terms of verification. The reason for this is that it relies on complex constructs such as constructors and destructors, templates, dynamic object allocation, virtual functions, and overload operators. These constructs cannot be represented directly in Promela and trying to represent their behaviour in a model can be very confusing and challenging.[25] Nasa in fact developed a C++ model checker known as the MCP model checker in order to verify flight control software for the crew exploration vehicle, Orion[25]. Although, unfortunately, we were unable to obtain this software. The MCP model checker uses a modified GCC front end to emit LLVM bytecode from a C++ source code file. It does not attempt extract a model from the code. The code is executed directly, executing all possible interleavings of transitions and states.[26]

We attempted to build a system to automatically translate C++ to a Promela model in a similar way. The intention was to find a method of automatically translating the C++ limbic system to a C source file, and using the Modex Tool, which will be described in section 3.4, to verify the generated C code.

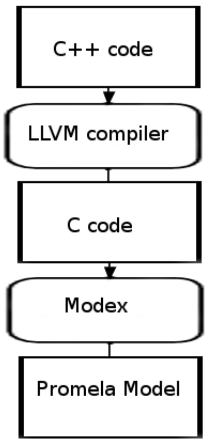


Figure. 8. Proposed architechure for a C++ to Promela translator.

3.2 Differences between C and C++

To fully understand the difficulty in creating a Promela translation of a C++ program compared to a C program, it is worth fully outlining the differences between the two languages. C++ retains almost all of C as a subset. The most notable difference between the two languages is that C++ allows for object-oriented design, where C does not.

In object-oriented design, we view a system as a collection of distinct objects, often based on real objects, whose behaviours may change over time. The state of an object is characterised by its state variables, and may be influenced through interactions with other objects.[16] We also the have ability to derive a class () from another (a class, often an abstract class) without knowing how its operations are implemented, and virtual functions in the case of C++. This provides a great deal extensibility and flexibility. This is relevant as Promela does not support object-oriented design, and so we must think very carefully about how to create an abstraction of a program. [1]

C is not a complete subset of C++ , as there are a few ways of expressing a computation in C that are not valid C++. This means that we cannot simply paste fragments of C++ code into a C source file and necessarily expect it to behave correctly.[1]

Some examples of this are given on Bjarne Stroustrup's C++ FAQ:

"Calling undeclared function in C is merely poor style in C, however it is illegal in C++

```
 int main()
{
    double sq2 = sqrt(2);    /* Not C++: call undeclared function */
    int s = sizeof('a');     /* silent difference:
                    1 in C++ sizeof(int) in C */
}
```

"[23] "C++ does not allow implicit violations of the static type system. For instance, in C, a void* can be implicitly converted to any pointer type. This is an issue because under these circumstances, if free-store allocation is done by using malloc(), there is no way of checking if enough memory is requested.

```
void* malloc(size_t);

void f(int n)
{
    int* p = malloc(n*sizeof(char)); /* not C++. In C++,
         allocate using 'new' */
    char c;
    void* pv = &c;
    int* pi = pv; /* implicit conversion of void* to int*.
                 Not in C++ */
}\cite{C++bjarne}
```

"

The correct way of doing this in C++ is to use the *new* operator

```
int* pi = new int;
```

[23]

C++ also has more keywords than C, meaning that well written C code will result in syntax errors if a keyword happens to appear somewhere in the code.

```
int class = 2; /* ok in C. Syntax error in C++ */
int virtual = 3; /* ok in C.
        Syntax error in C++ */
```

[1]

3.3 Automated Translation of C++ to C

Since it is not possible to directly translate C++ to Promela due to the advanced features of the language that Promela is not capable of representing, some research was done towards automatically translating C++ to C, with the intention of using Modex(See section 3.4)to verify the generated C code.

We thought it may be possible to automatically translate C++ code to C, as the original implementation of C++, *C with classes*, was built as an extension of the C programming language, and the original C++ compiler *Cfront* was capable of outputting C code from C++[12]. This code will not be maintainable since it is not intended to be human-readable. However, translated code may be possible to use as input for Modex to verify. There are a few tools that can perform this task:

Comeau Computing's Comeau C/C++ compiler has the capability of outputting C code form C++ input. Unfortunately it is a commercial product and requires purchasing a license to use. It may be worth investigating in future.

Cfront, the original compiler for *C with classes*. It can generate C code from older versions ofC++. We were unable to get a copy, and it does not support newer language features.

LLVM (Low-level virtual machine) is a commonly used compiler for Linux which, in the past, was able to directly output C code from C++. Unfortunately this feature is no longer supported as of version 3.0 due to lack of interest. Much effort was put into getting these older versions to run on more up to date versions of Debian, however we found that, given the scope of this project, it simply was not practical.

We attempted to use a modification for a more recent version of LLVM, which restores the C backend, however it too, had not been updated for some time and was incompatible with up to date operating systems.

Due to the lack of success in automatically translating C++ code to C, it is unknown at this point whether the generated C code would be suitable for verification by Modex, or whether it would be practical or even possible to understand the generated code well enough to write an effective test harness. Again, this may be something worth investigating in future.

3.4 Overview of Modex

Modex is a program which is used to extract verification models from C source code. This enables the verification of multithreaded C code, as well as the checking the basic computational properties of sequential code.[2] However according to its user manual, its strength lies in the former. A huge advantage of using Modex is that it doesn't require a detailed knowledge of model checking. However, it is good to understand how to write tests using linear time temporal logic and how to interpret the feedback of the Spin model checker. As the models generated from Modex can be hugely complex, unoptimised, and unreadable, Modex requires the definition of a test harness prior to any kind of model extraction. This test harness is where any tests the user wishes to apply to the model are defined, along with specific instructions as to the methods to be extracted. The necessity of this varies between different programs however. For instance, if the C source code is to be run with asynchronous concurrent threads of execution, the user must specify which functions this applies to.[2] A disadvantage of Modex is that knowledge of the C code and the hardware on which it runs(if any) has to be used in the abstraction process, because the model checker is not aware of this information.[24]

3.5 Testing Modex

To properly test the efficacy of using Modex to verify C code, we compared a manual translation of a C program to Promela with a Modex generated one.

So we take a simple program, workers, which is a simple multithreaded application. The program represents three workers with one hammer between them. When a worker has the hammer, he may swing the hammer down. Once he has done this he passes the hammer on to another worker, who will do the same.

The Manually written Promela representation of this is simple. The Hammer and its state are represented by the mtypes hammer, up, and down. Each worker is represented by a process and channel with space for one element. The contents of the channel will represent whether or not the worker has the hammer. The process has one mtype m, to which the data from the worker's channel is read. if worker m reads 'hammer' from the channel, the process jumps to the work section of the code, enabling the worker to swing the hammer down by setting the arm mtype to down. Once the hammer is swung down, arm is set to up the hammer mtype in non-deterministically written to either of the other worker's channels, allowing another worker to perform the same action. the non-deterministic rewriting of hammer is enclosed in an atomic statement.

A C implementation of this model was created using the pthread library for multithreading. Although the behaviour of the program is the same in each implementation, there are numerous differences in terms of implementation.

In place of the mtypes hammer,up and down, we use an enum hammerstate to represent the state of the hammer, the value of which can either be up or down.

It is defined as

```
typedef enum{up = (int)0,down = (int)1}hammerstate;
```

The workers are represented by the type

```
typedef struct worker{
    int id;
    int state;
} worker;
```

The integer *id* representing the *id* of the worker and the state representing whether the hammer is up or down. Three instances of the worker type, each with state set to 0 and a number assigned as the id. The main difference in the implementation is C's lack of channels, so instead of having hammer as an enum, hammer is a mutex lock from the pthread library. One pthread is initialised for each instance of the worker type. The function work in is run in each thread so that, in the same manner as the Promela implementation each worker has its own process. The area of the code where the hammer is swung down by a worker is protected by a critical section, and requires that the worker has the mutex hammer in order to execute it. Once the state variable is set to up, the mutex hammer is released and given to another worker process. The program output is printed to the console.Like with a Promela mtype, an enum in C is essentially an alias for an integer, so the program outputs the integer value for the enum hammerstate. In this run, worker 1 starts with the hammer.

```
step 0: Worker 1 swings the hammer down,  so value of worker 1 is 1 ,
     value of worker 2 is 0 , value of worker 3 is 0
step 0: Worker 1 lifts the hammer up and passes it to worker 3.
step 1: Worker 3 swings the hammer down,  so value of worker 1 is 0 ,
    value of worker 2 is 0 , value of worker 3 is 1
step 1: Worker 3 lifts the hammer up and passes it to worker 2.
step 2: Worker 2 swings the hammer down,  so value of worker 1 is 0 ,
    value of worker 2 is 1 , value of worker 3 is 0
step 2: Worker 2 lifts the hammer up and passes it to worker 1.
step 3: Worker 1 swings the hammer down,  so value of worker 1 is 1 ,
    value of worker 2 is 0 , value of worker 3 is 0
step 3: Worker 1 lifts the hammer up and passes it to worker 3.
step 4: Worker 3 swings the hammer down,  so value of worker 1 is 0 ,
    value of worker 2 is 0 , value of worker 3 is 1
step 4: Worker 3 lifts the hammer up and passes it to worker 2.
step 5: Worker 2 swings the hammer down,  so value of worker 1 is 0 ,
    value of worker 2 is 1 , value of worker 3 is 0
step 5: Worker 2 lifts the hammer up and passes it to worker 1.
step 6: Worker 1 swings the hammer down,  so value of worker 1 is 1 ,
    value of worker 2 is 0 , value of worker 3 is 0
```

and so on.

During the experimentational Modex translation of the multithreaded program ”workers.c”, it was also found that certain styles of declaring structures resulted in a syntax error in the pan.out file generated by Spin when verifying the model.

If the instances of the worker structs are initialised with variables in the following style:

```
worker worker0 = {1,0};
worker worker0 = {2,0};
worker worker0 = {3,0};
```

running

```
verify brokencworkers.c
```

results in the following error:

```
Extract Model:
==============
Modex brokencworkers.c
MODEX Version 2.10 - 25 June 2016
created: model and _Modex_.run

    Compile and Run:
    ================
    sh _Modex_.run
pan.c: In function  globinit :
pan.c:77:19: error: expected expression before  {  token
  now.worker0  =  {1, 0} ;
               ^
pan.c:78:19: error: expected expression before  {  token
  now.worker1  =  {2, 0} ;
               ^
```

```
pan.c:79:19: error: expected expression before   {   token
  now.worker2  =  {3, 0} ;
                ^

No Errors Found
```

This error is a result of the verifier generated by Spin containing syntax errors.

However if the instances of the worker struct are declared with no variables initialised like so

```
worker worker0;
worker worker1;
worker worker2;
```

and have their variables initialised by accessing the members like so:

```
worker0.id=1;
worker1.id=2;
worker2.id=3;

worker0.state=0;
worker1.state=0;
worker2.state=0;
```

the verification will run and reach a valid end state.

An issue with Modex and other automatic translators is the complexity of the models they produce compounded by the fact that the code is not designed to be read or maintained. When the manually written Promela model of the workers program is run, there are few states and the verification completes easily using little memory:

```
        State-vector 72 byte, depth reached 20, errors: 0
       33 states, stored
        9 states, matched
       42 transitions (= stored+matched)
        3 atomic steps
        hash conflicts:         0 (resolved)

        Stats on memory usage (in Megabytes):
    0.003       equivalent memory usage for states (stored*(State-vector + overhead))
    0.289       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage
```

The model generated from the C code by Modex was far less efficient than the manual translation. The test harness was also not particularly well documented and quite difficult to get working, and running without specifically labelling the concurrent *work()* functions, results in a model which is not only impossible to run, but not a faithful representation of the original program at all. Even when used correctly with a fully working test harness, the lack of options when it comes to abstraction result in a somewhat inefficient model, meaning that without access to a very powerful computer, the scope and complexity of the C code being checked is fairly limited compared to manual translation.

Taking into consideration that both of these models are of the same system, it is reasonable to infer that although a certain level of knowledge and skill is required to manually translate C to Promela, it is certainly advantageous to take that approach in terms of the quality of the final model.

Chapter 4

Planning the Manual Translation and Requirements gathering

4.1 Understanding the LSS

After the lack of success with the automated translation, we moved on to attempting a manual translation of the limbic system. The most challenging aspect of this approach was understanding the system itself.

This highlights one of the fundamental challenges of model checking: the barrier of understanding between people with different areas of expertise attempting to communicate, understand, and work based on, advanced concepts in the different fields. In this case, creating a faithful representation of the limbic system required us to understand some neuroscience concepts, and the different roles played by different parts of the brain. We were required to know how they are represented in the lss, and how those different components interact with one another. This was done using the process outlined earlier in Figure. 7.

While the whole modelling process was an iterative one of gaining a greater understanding of the simulation, for us it was also a process of better understanding of Spin and Promela. This compounded the difficulties with the modelling process as it created uncertainty as to whether errors in the model were a result of a genuine error in the original system, a lack of understanding of the original system, or simply a poor implementation on our part.

After numerous requirements gathering meetings with the developer of the lss, we were able to get a reasonable understanding of the system. This was done by making control flow diagrams detailing our understanding the various components of the system, getting feedback, refining the diagrams, and repeating the process. A diagram is shown in figure 7 in section 2.6.

4.1.1 Design of the lss

Below is a table containing the different classes of the lss and their functionality

Class	functionality
limbic_system	takes input signals from the agent's environment and produces output signals, directing the agent's movement.
filter	takes certain signal values from limbic_system and applies low and high pass filters to them.
robot	Definition of the agent which moves around the explorable area
direction	controls the direction in which the agent moves
world	Definition of the explorable area; controls switching of reward locations
limbic-system-simulator	Contains main method of lss. instantiates other objects and closes when simulation is complete
merge2quicktime	Controls graphical representation of the lss displayed in Quicktime

4.1.2 the limbic_system class

The *limbic_system* class is responsible for taking input signals based on the agent's environment and generating motor signals which direct the movement of the agent. As this is the class which contains the elements of the lss that we were interested in, and we were attempting to abstract the system using cone of influence reduction, we disregarded the other classes and only included components in the model that were necessary for the model to work as intended.

The limbic_system class contains two member functions which we are concerned with, *doStep* and *weightChange*.

The *weightchange* function is responsible for changing the value of particular based on the input, for example, the coreWeightLG2DG value is changed by adding $plasticity * visualDirectionLG * coreOutDG$ to its current value.

Below is the C++ of weightChange used in the lss:

```
void Limbic\_system::weightChange(float &w, float delta) {
     w += delta;
     if (w>1) w = 1;
     if (w<0) w = 0;
}
```

It takes the parameters $\&w$ a reference to the value to be modified and $delta$, the value by which we modify w. If the modified value w is greater than 1, or less than 0, then it set to 1 an 0 respectively.

doStep runs once for every step of the simulation. It takes as parameters a reward signal *reward*, a visual input signal for the light green landmark *visual_direction_DG* and for the dark green landmark *visual_direction_DG*, signals indicating when the agent is in the light green landmark *placeFieldLG* and the dark green landmark *placeFieldDG*, and a visual input when the agent sees the reward depending on its location *visual_reward_LG* and *visual_reward_DG*. All of the input signals are of type float. The values from these signals are used by the weightChange function to modify used to generate an output signal, and eventually a motor signal. The other values used in the doStep function are described in detail below.

mPFC_LG and *mPFC_DG*: The pre-frontal cortex. This value is used to smoothen signal changes and make the learning process more realistic and create memory traces so that they last until the agent finds the reward.

VTA: sum of the reward value and VTA_baseline_activity. Inhibited by the *RMTg*. Used to increase the *Shell_plasticity* and *core_plasticity*.

visualDirectionLG and *visualDirectionDG*: values based on the distance between the agent and a landmark in its field of vision (The light green landmark and the dark green landmark respectively). When the agent can see a landmark, the respective visual direction value will increase as the agent gets closer to it, and will decrease as the agent gets further away.

OFC: Orbital frontal cortex. This value represents the strength of association between a landmark and the reward value obtained by going there. It is based on two values called *pfLg2OFC* and *pfDg2OFC*, denoting an association between the reward and the light green landmark and the dark green landmark respectively.

RMTg: Used to inhibit the *VTA*. Same value as *EP* and *lHB*

DRN: Dorsal raphe nuclei. This value is based on the sum of the VTA and the reward value. Used to increase the strength of an *OFC* signal associated with a landmark.

lHB: Same value as RMTg.

EP: Inverse of the *dlVP*.

shell_DA: Same value as VTA.

lShell: Used to inhibit *dlVP*

dlVP: Used to inhibit the *EP*/shell_DA/*lHB* signals.

Shell_weight_pflg and *Shell_weight_pflg*: Used to change the strength of the *lShell* signal.

Shell_plasticity: Used to change the strength of *lShell_weight_pflg* and *lShell_weight_pfdg*. *core_plasticity*: Used to increase the core_weight_lg2lg, core_weight_lg2dg, core_weight_dg2lg and core_weight_dg2dg signals.

CoreOutLG and *CoreOutDG*: The motor signals used to direct the light green landmark and the dark green landmark respectively.

coreWeightLG2LG, coreWeightLG2DG, coreWeightDG2LG, and *coreWeightDG2DG*: The *output* signals which are the primary influence of the motor signals. Respectively They are based on a combination of the visual signals *visualDirectionLG* and *visualDirectionDG*, and the agents knowledge of the reward location(i.e the *visualRewardLG* and *visualRewardLG* variables). The signal with the highest value will determine which motor signal is used to move the agent. *step*: the current iteration of the doStep function. Their effect on the agent is shown in the table below:

signal	effect
coreWeightLG2LG	Agent sees light green landmark and wants to move to the light green landmark
coreWeightLG2DG	Agent sees light green landmark and wants to move to the dark green landmark
coreWeightDG2LG	Agent sees dark green landmark and wants to move to the light green landmark
coreWeightDG2DG	Agent sees dark green landmark and wants to move to the dark green landmark

A control flow graph for doStep is shown in figure 9.

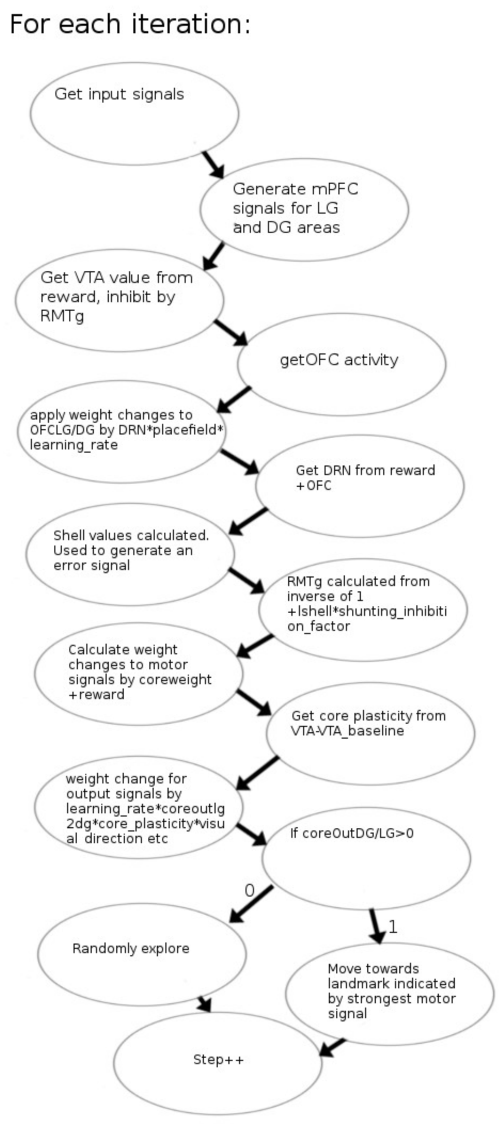

For each iteration:
Get input signals
Generate mPFC signals for LG and DG areas
Get VTA value from reward, inhibit by RMTg
getOFC activity
apply weight changes to OFCLG/DG by DRN*placefield* learning_rate
Get DRN from reward +OFC
Shell values calculated. Used to generate an error signal
RMTg calculated from inverse of 1 +lshell*shunting_inhibition_factor
Calculate weight changes to motor signals by coreweight +reward
Get core plasticity from VTA-VTA_baseline
weight change for output signals by learning_rate*coreoutlg2dg*core_plasticity*visual direction etc
If coreOutDG/LG>0
0
1
Randomly explore
Move towards landmark indicated by strongest motor signal
Step++

Figure. 9. Control flow graph for doStep.

To understand how the internal components used in doStep affect one another, we ran several simulations of the lss and observed its behaviour. This was done by inserting print statements into the program to see how signals changed depending on what the agent was doing in the simulation, and using a python script to generate graphs showing how the signals would change with respect to one another. The most recent graph is shown below:

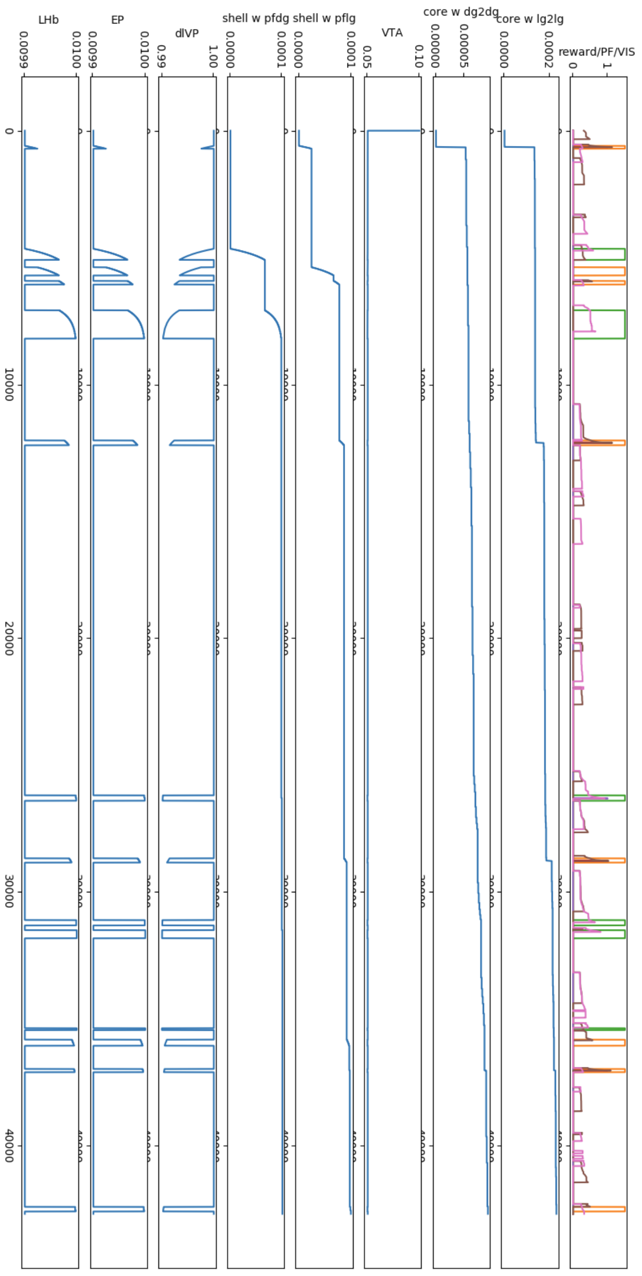


Figure. 10. Graphs showing how signals change over time.

From these graphs, we can gained reasonable insight into how the different signals can be expected to behave.

The agent's movement is controlled from within the *doStep* function. It's direction is determined by four floating point numbers *coreLGOut*, *coreDGout CoreExploreLeft*, and *CoreExploreRight*. If the agent can see a landmark and receives a visual input signal from it, and an output signal that puts one of the motor signals *coreLGOut coreDGOut* greater than or equal to 0.1, the agent will move straight towards that landmark. If *coreLGOut* and *coreDGOut* values are below 0.1, the agent's movement will be determined by the values *CoreExploreLeft*, and *CoreExploreRight*.

In this case, the code below is run. If the value of *CoreExploreLeft* is higher than *CoreExploreRight*, the agent will turn left as it moves forward, and right if *CoreExploreRight* is greater than *CoreExploreLeft*. If the values of the two variables are both the equal to one another and greater than 0, the agent will move straight forward. However, if they are both equal to 0, the agent will not move. Each iteration of the *doStep* function, the direction to be moved in the next iteration is randomly assigned to the variable *explorestate*. This means that when the agent will move randomly until one of the coreOut values are greater than 0.1. These output values are used by instances of the *robot* and *direction* classes to determine how the agent moves.

```
 switch (exploreState)
{
    case EXPLORE_LEFT:
        CoreExploreLeft = 0.1; // (float)random()/(float)RAND_MAX;
        CoreExploreRight = 0; //(float)random()/(float)RAND_MAX;
        if (((float)random()/((float)RAND_MAX))<0.05) {
            exploreState = (ExploreStates)floor((float)random()/
                        (float)RAND_MAX*EXPLORE_NUM_ITEMS);
        }
        //printf("left\n");
        break;
    case EXPLORE_RIGHT:
        CoreExploreLeft = 0; // (float)random()/(float)RAND_MAX;
        CoreExploreRight = 0.1; //(float)random()/(float)RAND_MAX;
        if (((float)random()/((float)RAND_MAX))<0.05) {
            exploreState = (ExploreStates)floor((float)random()/
                        (float)RAND_MAX*EXPLORE_NUM_ITEMS);
        }
        //printf("right\n");
        ...
        //cases EXPLORE_STOP and EXPLORE_STRAIGHT
    }
```

Chapter 5

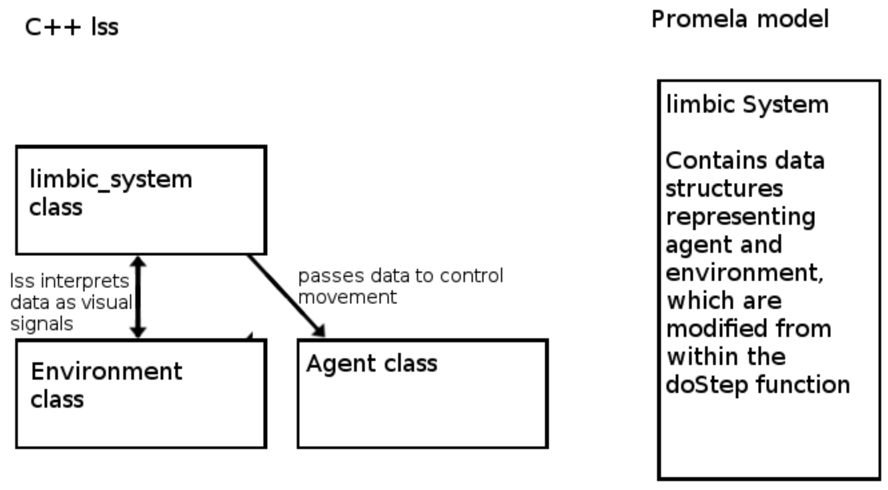
Modelling the Limbic System in Promela

5.1 Abstracting the lss

As discussed earlier, abstracting C++ program flow to Promela is very challenging due to the complex features of the language simply not existing in Promela. In this section we discuss how we were able to circumvent these problems by designing our model around those features.

Fortunately in the case of the lss, we did not need to deal with concurrency, templates, virtual functions, or multiple inheritance. No issues arose from inheritance.

The approach we used to circumvent the issues presented by the object-oriented design of the lss was to focus mainly on modelling the limbic_system class in fairly high detail, and to represent the other components of the lss, such as the environment and the agent itself as simple constructs within the model, as opposed to separate objects. The components of these constructs can be used as input data for our representation of the doStep function in a static context. The behaviour of these constructs was not based on the code itself, but instead on the results of rigorous observation and analysis of the behaviour of the agent in the environment. As the lss has no concurrency, this was relatively simple to do. These other classes could safely be simplified in our model, as their implementation is not relevant to the properties we wish to check. Below is a diagram showing a simplified representation of the class interaction of the lss, juxtaposed with the structure of the Promela model.


C++ lss
Promela model
limbic_system class
lss interprets data as visual signals
passes data to control movement
Environment class
Agent class
limbic System
Contains data structures representing agent and environment, which are modified from within the doStep function

Any issues that arise from this approach can be rectified in future by refining the model.

5.2 Expressing Floating Point Numbers in Promela

One interesting design element of Promela is the intentional omission of floating point numbers as a language feature. The reason for this is that Promela is designed to encourage abstraction from the computational aspects of an application, mainly so that the user can focus on process interaction,synchronisation and coordination.[4]

An example of how one should treat calculations which require floating points is given in the Spin user manual:

”Consider, for instance, the verification of a sequential C procedure that computes square roots. Exhaustive state-based verification would not be the best approach to verify this procedure. In a verification model, it often suffices to abstract this type of procedure into a simple two-state demon that non-deterministically decides to give either a correct or incorrect answer.”[4]

Take for example, the following simple C function which adds two floats together and executes a different function depending on the result.

```
int example(float a , float b)
{
    float c = a * b;

    if(c>10)
    {
        doFunctionA();
        return 0;
    }
    else{
        doFunctionB();
        return 0;
    }
}
```

Modelling this function using Promela as intended, we could simply omit the calculation on which the if statement is based and do the following:

```
proctype example()
{
    if
    ::doFunctionA
    ::doFunctionB
    fi
}
```

This would be a suitable representation of the example function. Because of the conditional statement, the program can take one of two paths. Either c will be greater than 10, and the program will execute doFunctionA; or c will be 10 or below and the program will execute doFunctionB. We can omit the calculation in the Promela implementation because the non-deterministic execution of the program means that both of these paths will be taken during verification. The actual calculation itself does not matter, only how the result is used to determine program flow.

This approach allows a program to be faithfully abstracted while being computationally simple enough to allow the construction of a runnable verification model.

Floats can be used in embedded C code, although, as mentioned in section 4.2, this creates a much larger number of potential states and is very computationally expensive to verify. When using embedded C code, the user can decide if the embedded data objects should be treated as part of the state descriptor in the verification model, with the use of c_state or c_track declarators. [4]

However, to model the lss to the level we desire, we needed to be able to represent floating point numbers in the model, while retaining as much computational simplicity as possible.

Alternatively, since exact values are generally not important, one could model numbers by using a number of discrete mtype values such as:

```
mtype = {min , low , medium , high ,  maximum}
```

This approach was also tried when attempting to model the doStep function. It turned out to be unsuitable for dealing with weight changes as it wasn't precise enough to deal with the difference between the output values, on which the behaviour of the agent is based. Specifically, as the weight change works by changing a value by a certain amount, we could find no effective way to determine that amount with enough precision to achieve the desired effect. However, this method was used for prototyping the weight change function. Many values in the lss are inhibited by dividing them by their inhibitor, and no satisfactory way of representing this was found while using this method.

So in order to achieve the compromise we needed between computational simplicity and precision, we implemented a simple system to simulate floating point numbers using a range of whole numbers. In order to manage the number of potential states that can arise from doing calculations based on large numbers, the precision of our simulated floating point numbers are determined by using the variable

```
byte precision;
```

The value of precision is used to represent the number 1. Shown below are the values represented by our model when precision is set to 100.

value in model	real value
100	1
10	0.1
1	0.01

We settled on using the value 100 after some experimentation with smaller and larger numbers. We found that using 100 gave a good balance between precision and computational simplicity.

Addition and subtraction using this method works perfectly well without needing any additional modifications, since for example, $100 + 367 = 467$ is analogous to $1 + 3.67 = 4.67$. However, with respect to multiplication and division, further steps are needed.

Using the default division and multiplication operators in Promela results in serious problems. Take for example the following multiplication: $0.25 * 0.25 = 0.0625$. Expressing this with our system as $025 * 025$ will result in the answer 625, which in our system is analogous to 6.25, not 0.625. The reason for this is that multiplying any number by a number less than 1 should result in a smaller product, which when using our system is not represented in the default multiplication operation.

Also, when using our representation, when we multiply two numbers greater than or equal to 1, we end up with a product which is far too large. For instance, using standard integer arithmetic, $1 * 1 = 1$, but in our system we end up with $100 * 100 = 10000$.

After noticing that these incorrect answers were out by a factor of the value of precision, we were able to create an inline procedure to tackle the problem.

```
inline multiply(x,y,out)// x * y, out is answer
{
    out = x*y;
    out=out/precision;

}
```

multiply takes three parameters, x and y and out, which are the numbers we wish to multiply, and the variable to which we would like to store the answer. We use the standard multiplication operator to get the product of the two numbers, and then divide the answer by precision. This results in the correct answer.

Using the default division operator presents problems along similar lines. For instance, if we take a representation of $2.5/0.5$ as $250/50$, the answers are the same: 5. This is incorrect because our system requires 5 to be represented as 500.

The solution to this problem is to divide with the inline function divide which takes as parameters: x as the numerator, y as the denominator, and out as the variable to which the answer will be saved. The numerator is multiplied by the precision value to achieve the correct answer.

```
inline divide(x,y,out)// x / y, out is answer
{

    out = (x*precision)/y
}
```

The reason the numerator is multiplied by precision before being divided by the denominator, instead of simply doing the division and then multiplying the answer by precision is since we are only using integers, we can at no point deal with any number < 1. Doing it this would mean that the value written to out would be incorrect.

5.3 Weight Changes

As described in section 4.1, the weightChange function in the lss which is responsible for modifying the values of components of the lss, and is instrumental in the learning process of the lss.

This was the first part of the lss that we attempted to model. The model was relatively simple to implement once we were able to adequately represent floating point numbers. To retain some of the structure of the limbic_system class, we represented the weightChange function as an *inline*.

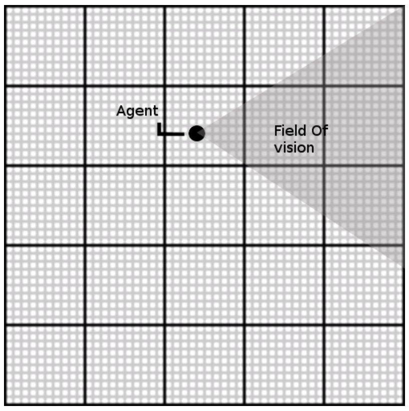
```
inline weightChange(w,d)//modify input value w by d.
{
    w=w+d;
    if
    ::(w>precision)->w=precision
    ::(w<0)->w=0
    ::(w>=0 && w<=precision)->skip
    fi
}
```

where *precision* is the value representing 1.

The model we developed behaves in almost exactly the same way as the lss. The only exception being that in the C++ implementation, a reference to w is passed as a parameter, rather than w itself. The reason for this is that C++ passes parameters by value, meaning that if we pass a variable as a parameter to a function, the value of that variable is given to a copy of the variable to be used in the scope of the function. As a result, any modifications made to that value only apply to the new variable instantiated inside the function, rather than the original variable we passed in. In C++, we need to explicitly specify when we want to modify a variable within a function by passing a reference to that variable.[23] In Promela, we do not need to do this as Promela always passes parameters by reference.

5.4 Abstracting the Area and Agent Movement

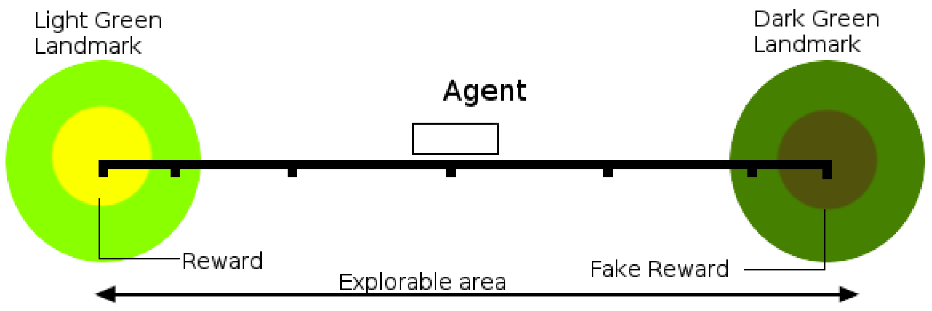
Representing the agent moving around the area was a challenging design problem. We needed to implement both the agent and the explorable are, both of which are classes independent of limbic_system in the lss. In addition to this, in the lss, the agent has a large area to explore. The number of positions that the agent can occupy, compounded by the number of orientations it can have yields an extremely large number of states to represent. To give an idea of how much this would complicate the model and impede performance, suppose that the agent can only move in a very small area of 50 by 50 positions and that it can face one of 360 directions.



This would result in $50 * 50 * 360$ states for the position and orientation of the agent alone, which is less than ideal. We had to come up with a more optimised solution for representing the agent's movement and position.

With respect to the limbic system, there are only a few properties relating to the position of the agent that are of any importance: whether it can see the landmarks and which landmarks in can see, and its distance from those landmarks; whether or not the agent is in a landmark; and whether or not the agent can see a reward;

We designed a way of representing the agent in the area which only represents these properties. The idea is two have the agent move along a short unidimensional line.with some empty space and the landmarks and the real and fake rewards. The agent was to be able occupy one of seven points on this line and move between those points as necessary.



The agent's vision can be in one of four states. It can see the light green landmark, the dark green landmark, both landmarks, or neither landmark. When the agent can see a landmark, it can determine the distance to that landmark.

This was implemented by defining *mtypes* to represent the different visual inputs and different parts of the area and.

```
mtype = {seeLG,seeDG,seeNothing,seeBoth,rewardObject,
         noRewardObject,areaLG,areaDG,nothing}
```

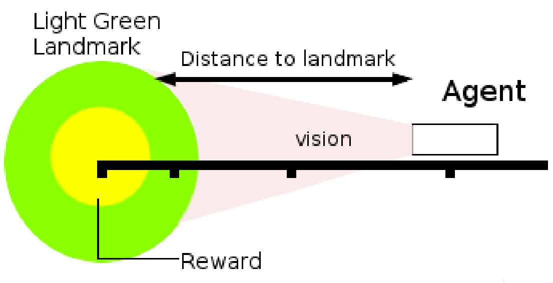
where *seeLG*, *seeDG*, *seeBoth* and *seeNeither* are assigned to the variable *vision* to represent the agent seeing the light green landmark, seeing the dark green landmark, seeing both landmarks and neither landmark respectively; and where *rewardObject* represents the reward, *noRewardObject* represents the fake reward. *areaLG* represents the light green landmark, *areaDG* represents the dark green landmark and *nothing* represents empty space in the area.

The area itself is represented a one-dimensional array, and a byte agentPos is used to represent the position of the agent in the array.

```
byte grid[7] = {rewardObject,areaLG,nothing,nothing,
nothing,areaDG,noRewardObject};
```

if agentPos is greater than the index of areaDG, or less than the index of areaLG, the agent is considered to be within the light green and dark green landmarks respectively. There are three indexes containing *nothing* in order to represent different distances from either landmark.

For the agent to determine its distance from a landmark, it must be able to see the landmark. The displacement between the agent's index and the landmark's index is taken as the distance.



The following code fragment is used to get the distance from a the light green landmark and uses it to determine the value of *visualDirectionLG*, if it can see the light green landmark.

```
if
::vision==seeLG->
    visualDirectionDG=0;
    if
    ::(agentPos==2)->visualDirectionLG=1;
    ::(agentPos==3)->visualDirectionLG=2;
    ::(agentPos==4)->visualDirectionLG=3;
    ::else->skip
    fi

...
//

fi
```

If the agent is in a landmark, its vision will always be set to only see that landmark. This step was taken to simplify the behaviour of the agent in order to focus more on the internal workings of the limbic system. At this stage, the agent can only see the reward if it is within the same landmark as the reward, and if it sees the landmark, it will always consume it. This is the best approximation of having the agent stop and wait for the landmark to turn red before it is consumed that we were able to develop in the given time. Although it will be fairly straightforward to implement this behaviour, it will have to be done in future. This will be discussed further in the future work section.

If the agent consumes either of the reward objects, agentPos will be reset to 3, i.e. the middle of the array, and its vision will be set so that it sees both landmarks, as is the case in the original system.

The agent is required to be able to randomly explore the area when its motor signals are low enough so as not to direct it to a landmark. This is done using Promela's non-deterministic execution to move the agent up along the by one index in a random direction. *vision* is also non-deterministically reassigned with one of the mtypes representing field of vision.

Random exploration is done with the following inline:

```
inline randomExplore()//randomly explore area
{
if
::vision=seeNothing
::vision=seeBoth
::vision=seeLG
::vision=seeDG
fi
if
::(grid[agentPos+1]!=noRewardObject && grid[agentPos+1]!=rewardObject)
    ->agentPos=agentPos+1
::(grid[agentPos-1]!=rewardObject && grid[agentPos-1]!=noRewardObject)
    ->agentPos=agentPos-1
fi
}
```

When the value of a motor signal is high enough for intentional movement to occur, the agent will move in the direction indicated by the motor signal which is higher.

Intentional movement is done with the following inline:

```
inline moveAgent() //intentional movement of agent
{
if
::(grid[agentPos]!=areaLG || grid[agentPos]!=areaDG)->
    if
    ::(coreLGOut>coreDGOut)->agentPos=agentPos-1;vision=seeLG
    ::(coreDGOut>coreLGOut)->agentPos=agentPos+1;vision=seeDG
    fi
fi

}
```

where *coreLGOut* and *coreDGOut* are the motor signals which direct the agent to the light green landmark and dark green landmark respectively.

The movement of the agent around the area is done between each iteration of our the doStep function in the model. All of the data structures and variables used to represent them are global in scope. This was the best way of modelling the agent and the area as though they were independent objects.

Every 5000 iterations of doStep, we switch the location of the reward and fake reward using the *switchreward* inline:

```
inline switchReward()//swap location of reward and fake reward on step n
{
if
::(step==5000)->
    if
    ::(grid[0]==rewardObject)->grid[0]=noRewardObject;grid[6]=rewardObject;c=c+1;
    ::(grid[0]==noRewardObject)->grid[6]=noRewardObject;grid[0]=rewardObject;c=c+1;
    fi
    step=0;
::(c==3)->goto end;
::else->skip
fi
}
```

After the reward has switched places 3 times, the simulation ends.

5.5 Modelling the doStep function

Using our representation of floating point numbers, we were able to model the calculations in the doStep function in very high detail. The calculations are almost exactly the same as in the original lss. There are a few values that are omitted however, namely the *mPFC* signals which are used to smooth out signals. The reasons for this is twofold: we wanted to limit the number of potential states wherever possible, and based on our observation of the lss, the effect of these values appeared to be minimal; and we were unable to reach a point where we were confident enough in our understanding of the effect of the filter class on the system. We expect that in future, we will be able to gain a suitable understanding of these signals and the role they play, and efficiently incorporate them into the model. We also made shell_plasticity and core_plasticity into one value *plasticity*, since they both have the same value.

We apply weight changes to different signals in exactly the same way as in the lss, by calling the weightChange inline. At the end of each iteration of doStep, we move the agent either by, in the case of the agent using its intentional movement, calling the moveAgent() inline, and in the case of random movement, the randomExplore inline

The only issue we found with using our floating point representation in the doStep function is one of readability. In many cases in the lss, the weightChange function required a $delta$ value which is a product of other values. For example, when weightChange is applied to core_weight_lg2lg:

```
weightChange(core_weight_lg2lg, learning_rate_core
        * core_plasticity * visual_direction_LG * CoreLGOut);
```

Since we are using 100 to represent 1, it is not sufficient to simply multiply learning_rate_core, core_plasticity, visual_direction_LG, and coreLGOut. We need to use our *multiply* function, which, at this stage, is only capable of taking two parameters. As a result, we cannot represent this entire multiplication in a single call of the *multiply* function.

Our solution to this was to instantiate two new local integer variables, each one to be used for the output of a single call of *multiply*.To improve readability, we adopted a strict naming scheme for these variables, for example if we were to multiply *value1* and *value2*, the output variable would be named *value1xValue2*.

The aforementioned readability issue comes from the fact that for a single call of *weightChange*, several of these local variables and several calls of *multiply* are needed for a single call of *weightChange*. For example, in the case of the weight changes of the core weight variables responsible for directing the intentional movement of the agent (*core_weight_lg2lg, core_weight_lg2dg etc*), this adds up to a large amount of code for a few simple calculations:

```
int plasticityxVisualDirLG;
int plasticityxVisualDirLGxcoreLGOut;
int plasticityxVisualDirLGxcoreDGOut;
int plasticityxVisualDirDG;
int plasticityxVisualDirDGxcoreLGOut;
int plasticityxVisualDirDGxcoreDGOut;

multiply(plasticity ,visualDirectionLG ,plasticityxVisualDirLG);
multiply(plasticityxVisualDirLG ,coreLGOut, plasticityxVisualDirLGxcoreLGOut);
multiply(plasticityxVisualDirLG ,coreDGOut, plasticityxVisualDirLGxcoreDGOut);

multiply(plasticity ,visualDirectionDG ,plasticityxVisualDirDG);
multiply(plasticityxVisualDirDG ,coreLGOut, plasticityxVisualDirDGxcoreLGOut);
multiply(plasticityxVisualDirDG ,coreDGOut, plasticityxVisualDirDGxcoreDGOut);

weightChange(coreWeightLG2LG, plasticityxVisualDirLGxcoreLGOut);
weightChange(coreWeightLG2DG, plasticityxVisualDirLGxcoreDGOut);
weightChange(coreWeightDG2LG, plasticityxVisualDirDGxcoreLGOut);
weightChange(coreWeightDG2LG, plasticityxVisualDirDGxcoreDGOut);
```

This issue could likely be solved by some refactoring.

An automata of the complete model is shown below in figure 11.

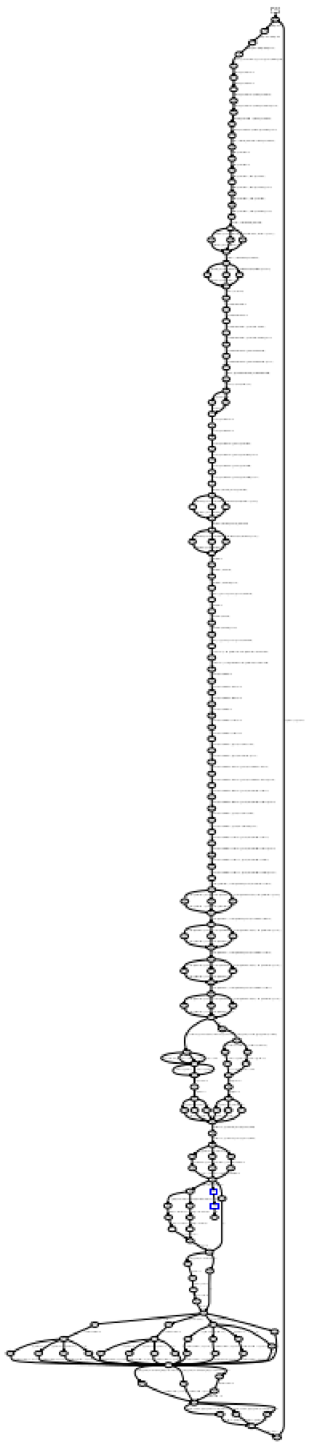


Figure. 11. Automata of the limbic system model.

Chapter 6

Verifying Properties of the Limbic System

6.1 Tuning the Model for Verification

Before we attempted to verify the model, we needed to tune to make it verifiable.

The guard in the *switchReward()* inline needed to be set so that it was high enough for the agent to generate a steady output signal directing it to the landmark containing the reward, but also low enough to limit the number of states and transitions.

As the program will go to the end state after the reward has switched places 3 times,the value of n in $(step == n)$, where n is the number of iterations of the *doStep* function since the reward last switched places, determines how long the program will run for.

```
inline switchReward()//swap location of reward and fake reward on step n
{
if
::(step==10000)->
     if
     ::(grid[0]==rewardObject)->
          grid[0]=noRewardObject; grid[6]=rewardObject;c=c+1;
     ::(grid[0]==noRewardObject)->
          grid[6]=noRewardObject; grid[0]=rewardObject;c=c+1;
     fi
     step=0;
::(c==3)->goto end;
::else->skip
fi

}
```

6.2 Invalid End States

The first verification run was to check the model for invalid end states. This was done by generating a verifier, *pan.c*, from the model.

```
spin -a limbicSystemAbstract.pml
```

Next we compiled the verifier, setting the maximum amount of memory that can be dedicated to the verification as well as ... . Due to the complexity of the model we allocate around 300 gigabytes of memory to the verification.

```
gcc -DMEMLIM=307200 -O2 -DXUSAFE -DNOCLAIM -w -o pan pan.c
```

Finally, we performed the verification my running the verifier *pan*, specifying the maximum depth of the depth-first search graph that will be generated with the *-m* flag. We set the depth to a very large number to account for the complexity of the model.

```
./pan -m10000000
```

Upon completing the verification run, we see that there are no errors in the model with respect the criteria we have defined. The program is always able to reach a valid end state.

```
State-vector 248 byte, depth reached 619198, errors: 0
2.960361e+08 states, stored
  7519840 states, matched
3.0355594e+08 transitions (= stored+matched)
```

6.3 Verification with LTL Properties

As we described in chapter 1, our objective was to discover if the agent can always learn the location of a reward, and unlearn the location of a reward once it has has been learned.

We define the agent as having learned the location of a reward it uses its intentional movement to move to the reward location, and having unlearned it by reverting back to random exploration.

We started by checking if *moveAgent*, the function which moves the agent intentionally once it has sufficiently high output signals, is always able to be used in a simulation run.

To do this we added a boolean valiable *usingCoreOut* to the moveAgent function. *usingCoreOut* is set to true if the agent is moving using moveAgent, and false if it is not.

We represent our correctness property as the LTL formula $<> p$, where p is true when *usingCoreOut* equates to $true$.

In English, this can be represented as follows:

For every path, p will ***eventually*** *be true*

This was represented in the model by adding the line

```
#define p (usingCoreOut==1)
...

ltl {<>p}
```

We then regenerated and recompiled the verifier in the same ways as described in section 6.2, and ran it with the accept labels flag $-a$:

```
./pan -m10000000 -a
```

Testing the model against the LTL formula resulted in an error. A path existed in the model where a path of was discovered where usingCoreOut is never equal to 1.

We printed out the error trail to tried and identify the cause of the error.

```
spin -p -t limbicSystemAbstract.pml
```

Our analysis of these results is in section 6.4.

We then ran an experiment to check if the agent can unlearn the location of a reward once it has has been learned.

In terms of our model, we wanted to check if *moveAgent*, the function which moves the agent intentionally towards a landmark, will always eventually be followed by *randomExplore*, which is used to move the agent randomly when the output signals are too low to trigger intentional movement. The only exception to this would be if the program reaches its end state when it is moving intentionally, in which case . To do this, we used an additional boolean variable, *usingRand*, which is set to true if randomExplore is called, and false if randomExplore is not being used to move the agent.

We represent our correctness property as theLTL formula $[](p \implies <> q)$, where p is true when *usingCoreOut* equates to $true$, and q is true when *usingRand* equates to $true$.

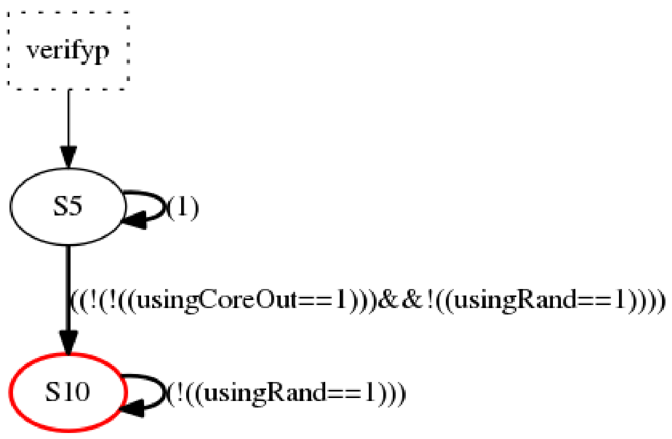
In English, this can be represented as follows:

*For every path, from **any** state, p implies that q will **eventually** be true*

We specified this property in the Promela model by adding the line

```
ltl {[](p -> <>q)}
```

The automata view in *ispin* displays this claim as shown below:



This means that when the verifier is run, it will check to see if the model satisfies the LTL properties.

Testing the model against the LTL formula resulted in an error. A path existed in the model where a path of was discovered where usingCoreOut being equal to 1 is never followed by usingRand being equal to 1.

Again printed out the error trail to tried and identify the cause of the error.

```
spin -p -t limbicSystemAbstract.pml
```

6.4 Analysis

6.4.1 Checking if the agent can always eventually learn

Looking at the path executed in the error trail, we noticed that because the agent's initial exploration is non-deterministic, there exists a possibility that it will never randomly enter a landmark and get the reward. Hence, it will be unable to build any association between the reward and the location and subsequently generate output signals compelling it to return to the landmark.

6.4.2 Checking if the agent can always eventually unlearn

Looking over the error trail to try and determine the cause of the error we found that the end state of the program was a valid end state where the agent's motor signal coreLGout had not dipped low enough for it to unlearn the location of the reward. The value in question, *coreLGOut*, was considerably higher than expected, compared to the values observed in the lss. This alone suggests the possibility of a serious disconnect between the behaviour of the original lss and our model of the lss, which will have to be remedied in future. Specifically, the problem seems to be a result of the number of *doStep* iterations between the reward changing places being too small to accommodate this large a value dropping far enough. While it is possible that we have discovered a remote error that exists in the original lss, at this stage the model of the limbic system simply may not be a close enough representation of the lss to draw any solid conclusions. If it is indeed the case that this is simply a problem with the model.

Chapter 7

Conclusions

Our initial objective was to analyse an agent-based learning system implemented in the C++ programming language and check whether or not the agent can always learn to find its reward rather than simply randomly find it by chance, and whether or not it can unlearn the location of a reward. While the model of the limbic system is somewhat unrefined at this stage, we are able to definitively state that, with some creativity with regard to abstraction, it is possible and practical model an agent-based learning system implemented in C++, such as the lss, using Promela to be verified with the Spin Model Checker. We found that the most practical way was to do so manually by gaining a thorough understanding of how the system operated by studying the code and observing its behaviour; and using cone of influence reduction to create an efficient and precise model.

Although the agent was unable to always eventually learn to find the reward, and unable to subsequently unlearn its location, we did discover the source of the errors. At this stage, it is possible that we have discovered errors that genuinely exist in the lss, it is probable that our model needs further refinement. In this case, the results of the experiments should be useful in achieving a better model.

7.1 Future work

This project presents some interesting opportunities for future work.

7.1.1 Refining the Model

As the process of developing the model of the lss is an iterative one, the model will need significant refinement if it is to faithfully model the original lss in order to effectively verify it, and be able to state with certainty that the lss works or does not work as intended. This can be done by continuing the process shown in section 2.6.

7.1.2 Automatic Translation

While we encountered problems creating a system to automatically translate of C++ to Promela, it would still be worth investigating into the future, as it has been done successfully in the past.

7.1.3 Investigate Modelling Multithreaded C++

The lss did not incorporate multithreading. It would be interesting to investigate the efficacy of modelling a multithreaded system implemented in C++.

7.1.4 Investigate Modelling C++ code containing virtual functions, templates,

Many of the advanced constructs that can be used in C++ were absent in the lss. It would be interesting to try and implement a Promela model of a C++ program which incorporates some advanced features.

Bibliography

[1] Bjarne stroustrup c++ faq, archived 2008. `https://web.archive.org/web/20080617183013/http://www.research.att.com/~bs/bs_faq.html`. Accessed: 26-03-2018.

[2] Modex user manual. `http://spinroot.com/modex/MANUAL.html`. Accessed: 27-03-2018.

[3] Promela user manual: Arithmetic and boolean operator precedence rules, high to low. `http://spinroot.com/spin/Man/operators.html`. Accessed: 20-03-2018.

[4] Promela user manual: Floats. `http://spinroot.com/spin/Man/init.html`. Accessed: 20-03-2018.

[5] Promela user manual: If. `http://spinroot.com/spin/Man/if.html`. Accessed: 28-03-2018.

[6] Promela user manual: Init. `http://spinroot.com/spin/Man/init.html`. Accessed: 20-03-2018.

[7] Promela user manual: Integer types. `http://spinroot.com/spin/Man/datatypes.html`. Accessed: 18-03-2018.

[8] Promela user manual: Proctypes. `http://spinroot.com/spin/Man/proctype.html`. Accessed: 20-03-2018.

[9] What is spin? `http://spinroot.com/spin/whatispin.html`. Accessed: 20-03-2018.

[10] Technical support to the national highway traffic safety administration(nhtsa) on the reported toyota motor corporation(tmc) unintended acceleration(ua) investigation. `http://spinroot.com/spin/success.html`, 2011. Accessed: 25-03-2018.

[11] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer-Verlag, London, 2008.

[12] Kent Budge, James S. Peery, Allen C. Robinson, and M Wong. Management of class temporaries in c++ translation systems. 06 1999.

[13] Orna Grumberg Edmund M. Clarke and Doron Peled. *Model Checking*, chapter 1, pages 2–3. The MIT Press, Cambridge, Massechusetts, 1999.

[14] Orna Grumberg Edmund M. Clarke and Doron Peled. *Model Checking*, chapter 1, page 1. The MIT Press, Cambridge, Massechusetts, 1999.

[15] Orna Grumberg Edmund M. Clarke and Doron Peled. *Model Checking*, chapter 13, pages 193–194. The MIT Press, Cambridge, Massechusetts, 1999.

[16] Gerald Jay Sussman Harold Abelson and Julie Sussman. *Structure and Interpretation of Computer Programs*, chapter 13, pages 193–194. The MIT Press, Massachusetts Institute of Technology,Cambridge, Massechusetts, 1985.

[17] Gerard J.Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):–, 1997.

[18] Pim Kars. The application of promela and spin inthe bos project(abstract). `http://spinroot.com/spin/symposia/ws96/Ka.pdf`, 1996. Accessed: 26-03-2018.

[19] Ryan F. Kirwan. *Applying Model Checking to Agent-Based learning Systems*. PhD thesis, University of Glasgow, 2014.

[20] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), September 2006.

[21] Pablo Moreno-Ger, Rubn Fuentes-Fernndez, Jos-Luis Sierra-Rodrguez, and Baltasar Fernndez-Manjn. C++ for safety-critical systems. *Information and Software Technology*, 51(3):564 – 580, 2009.

[22] Daniel Ratiu and Andreas Ulrich. Increasing usability of spin-based c code verification using a harness definition language.

[23] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*, page 145. Addison-Welsley, Reading, Massachusetts, 1997.

[24] MODEL CHECKING C SOURCE CODE FOR EMBEDDED SYSTEMS. Bastian schlich and stefan kowalewsk.

[25] S. Thompson and G. Brat. Verification of c++ flight software with the mcp model checker. In *2008 IEEE Aerospace Conference*, pages 1–9, March 2008.

[26] Sarah Thompson, Guillaume Brat, and Karl Schimpf. The mcp model checker.

[27] Robert W.Floyd. Non-deterministic algorithms. *Journal of the ACM*, 14, November 1967.