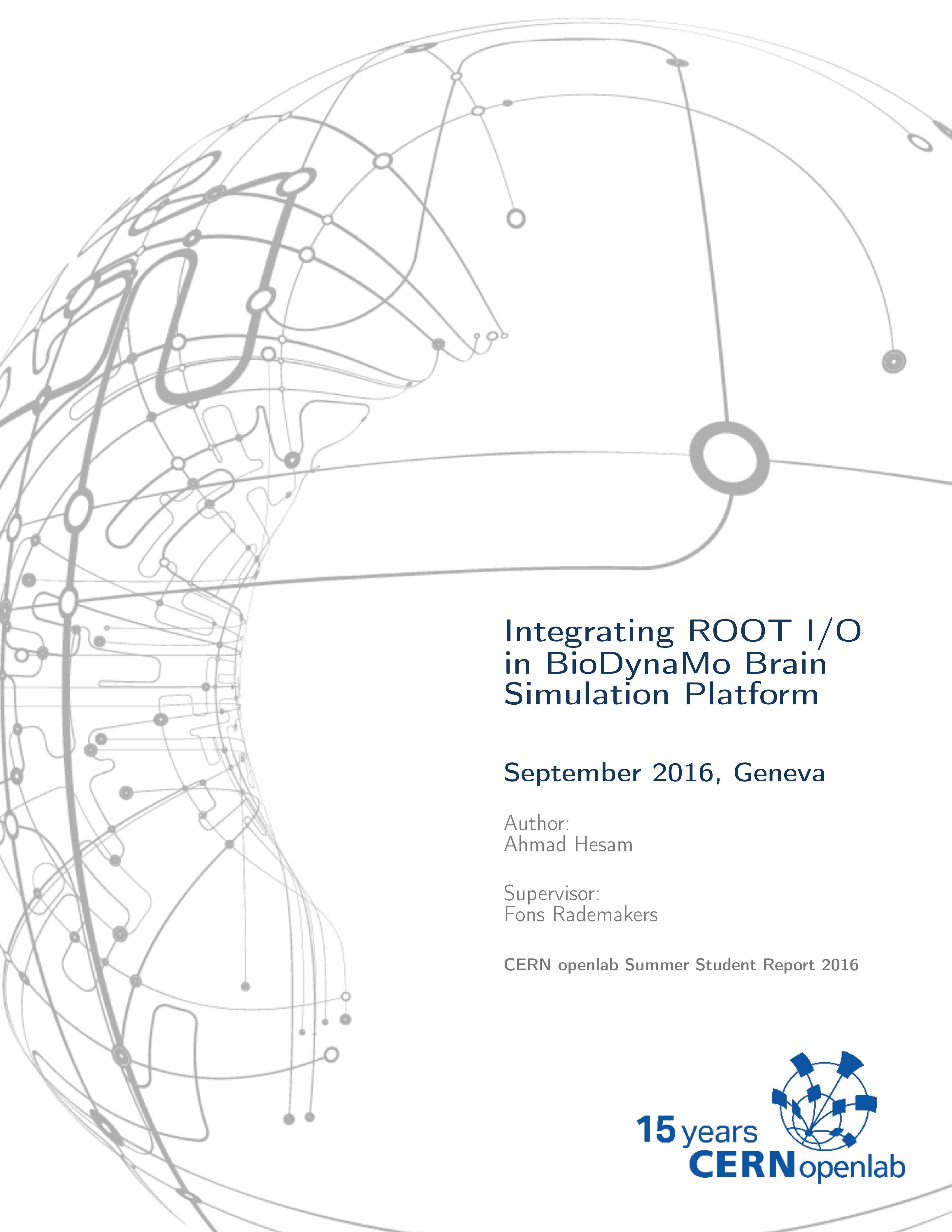

Integrating ROOT I/O in BioDynaMo Brain Simulation Platform

September 2016, Geneva

Author:
Ahmad Hesam

Supervisor:
Fons Rademakers

CERN openlab Summer Student Report 2016

Project Specification

In this project a brain simulation code has been ported from Java to C++ (BioDynaMo). The porting is now finished and the project goes into the next phase of code optimization and ROOT (`https://root.cern.ch`) integration. The summer student's task is to integrate ROOT I/O technology to make the simulation output persistent, so that simulations can be stopped and restarted. This is important to be able to snapshot very long running simulations to not lose many hours or days of work in case the program crashes or a machine goes down. In addition, I/O is important to be able to distribute simulation state between nodes running one large simulation in parallel. ROOT I/O is the key I/O technology used to store all scientific experimental data at CERN and in other HEP labs. The student should be good in C++ programming and will learn how advanced cell growth codes work and how they can be used to simulate certain common diseases.

Fons Rademakers

Acknowledgements

First and foremost, I would like to thank my supervisor Fons Rademakers for giving me the chance to work on this project and for the technical support. Secondly, I want to thank Lukas Breitwieser for his time and effort in helping me with debugging and coding issues. Also a big thanks to the ROOT development team for their quick replies and insights concerning ROOT for this project; Philippe Canal, Danilo Piparo and Axel Naumann. And lastly, I want to thank the CERN openlab team and students for making the summer of 2016 an unforgettably great experience.

Abstract

A brain simulation platform called BioDynaMo has been made data persistent using the I/O features of CERN's Big Data software framework ROOT. During runtime of a simulation the entire simulation state can be written to file and read back whenever required. This result preserves computational resources, because in case of a crash in simulation runtime, a persisted simulation state can be reloaded in memory, instead of repeating the simulation up till the crash. For several simulations it has been measured that the overhead for writing a simulation state to file is between 1.0 - 1.4 seconds. Persisting the state every half hour makes this overhead negligible. The files generated to hold the persistent state is sized between 6 kB - 1.6 MB, which is up to 15× smaller than files in JSON format holding the same information. A data persistent BioDynaMo also allows for distributing the simulation state over multiple computer nodes that run the simulation in parallel. This however still needs to be explored in more depth.

Table of Contents

1 Introduction **5**

2 The BioDynaMo Project **6**
2.1 A Brain Simulation Platform . . . 6
2.2 Persistent Simulation Data . . . 6
2.3 Development environment . . . 7

3 ROOT I/O **8**
3.1 The Essentials . . . 8
3.1.1 Streamers . . . 8
3.1.2 ClassDef Macro . . . 9
3.1.3 Dictionaries . . . 10
3.1.4 LinkDef File . . . 11
3.2 Data Persistence . . . 11
3.2.1 TFile Object and ROOT File . . . 12
3.2.2 WriteObject() and GetObject() . . . 13

4 Making a Class Persistent **14**

5 Workflow **16**
5.1 ROOT I/O in BioDynaMo . . . 16
5.2 Persisting and Building . . . 17
5.3 Testing . . . 18
5.4 Debugging . . . 19
5.5 Profiling . . . 19
5.6 Current Status and Future Work . . . 20

6 Troubleshooting **22**

Appendix A **26**

Appendix B **28**

1 Introduction

One of the most, if not the most, popular scientific research question of today is why the human brain works the way it works. In 2013 the European Union launched a flagship project under the name The Human Brain Project (HBP), which financially aids researchers and institutions in seeking knowledge on the human brain in the fields of neuroscience, computing, and brain-related medicine. In addition, other institutions are making efforts for the same purpose as the HBP, including CERN.

CERN has several on-going externally funded projects, of which one is the Human Brain Development project, which is separate from the HBP. This project aims to optimize the code of the Cx3D brain simulation software framework that was developed at ETH Zürich. Lukas Breitwieser, who was a CERN openlab student in 2015, ported this code from Java to C++ and set up a solid development environment [1]. The platform based on the Cx3D framework is now called BioDynaMo. There are several future plans to improve the platform. One of them is the integration of ROOT I/O in the simulation framework in order to make the simulation data persistent. ROOT is CERN's Big Data software framework that includes an API (ROOT I/O) that enables data persistence through data serialization. Data persistence is useful in order to snapshot very long running simulations to not lose hours or days of effort whenever the program or a machine crashes. Additionally data serialization is important in order to create a distributed simulation state between nodes running a single simulation in parallel.

In this report the implementation of ROOT I/O into the BioDynaMo platform is described. It includes an overview on the BioDynaMo platform, the basics of ROOT I/O, and an explanation on how BioDynaMo classes were made persistent using ROOT I/O. The problems that were faced during the implementation will be discussed as well as their solutions.

2 The BioDynaMo Project

The goal of BioDynaMo is to provide life scientists a scalable, yet flexible platform to simulate brain activities. This chapter will give an introduction to BioDynaMo and emphasize the importance of data persistence.

2.1 A Brain Simulation Platform

The BioDynaMo platform is based on the simulation program Cx3D developed by the University of Zürich and ETH Zürich, which can simulate the development of neural networks such as the neocortex [2]. The aim of BioDynaMo is to be able to simulate, in a more general sense, biological tissue dynamics, which includes brain development. Cell proliferation and neurite growth are examples of biological tissue dynamics that can currently be simulated. The two main aspects BioDynaMo distinguishes itself from other simulation platforms are the heuristic approach to simulation and the compactness of the code. A heuristic is a rule defined by the user, which for example lets neurites grow in the direction of a specific extracellular substance gradient. The code base is currently 15 kLOC, which is very compact and allows for convenient exploration of platform scalability.

Simulating the aforementioned instances of biological tissue dynamics is useful in many scientific fields that are not directly related to the Human Brain Development Project. Neuroscience, cancer research, plant growth and tissue growth are biologically relevant topics that BioDynaMo will aim to become the standard platform for. In order to meet this goal it is of vital importance that there are sufficient computational resources to support the platform. More importantly, the integration of BioDynaMo on the resources should be as effective and efficient as possible to provide scientists a flexible and scalable simulation platform.

2.2 Persistent Simulation Data

Simulating brain activities requires immense computational resources. In [3] it was tested that a single batch simulation of a network of 100 neurons, reproducing three weeks of neuron growth, with a simulation step of 0.1 hours, could take Cx3D up to five weeks to complete on a single PC. If the simulation or PC would crash near the end of completion, days or weeks of computational resources would be wasted.

One way to preserve computation resources is to have a data persistent simulation. For a simulation, data persistence means to create periodical snapshots of the entire simulation state and store them as files on disk. Whenever a crash would occur, a snapshot that was taken before the crash can be reloaded into the memory of the system and the simulation can be continued from that state onwards. A trade-off needs to be made between simulation performance and frequency of persisting a state, as undoubtedly there will be overhead introduced when creating a snapshot of the simulation state.

2.3 Development environment

The source code of BioDynaMo can currently be found on Github [1], where an international team of computer scientists are contributing to the platform. In order to have an efficient workflow in which quality code is produced it is important to have a well-defined development environment. In the BioDynaMo Developer's Guide [2] one can find the rules and practices that contributing developers should hold onto.

[1] https://github.com/BioDynaMo/biodynamo

[2] https://github.com/BioDynaMo/biodynamo/wiki/BioDynaMo-Developers-Guide

3 ROOT I/O

ROOT is a scientific software framework used by CERN to analyse and visualize data, such as experimental high-energy physics data. Part of this framework is for input and output (I/O) of data from memory to disk and vice versa. This chapter will outline the essentials of ROOT I/O and how it enables data persistence. The assumption is made that the reader is familiar with the basics of Object Oriented Programming (OOP).

3.1 The Essentials

ROOT is written in C++, an OOP language. Whenever an object is instantiated during the runtime of a computer program, memory is allocated to facilitate for its existence. An object may have many data members, which could be other objects, pointers to other objects or any complex composition of objects in objects. ROOT I/O is able to find data members in memory that are of interest to the user and serialize them to system-independent binary files (supporting Big Endian byte ordering, ASCII, IEEE floating point). These files, called ROOT files, can be stored on disk and can be deserialized and loaded into memory whenever the user wants.

3.1.1 Streamers

Data serialization is the key process to ROOT I/O. It is the process of translating an object in memory into a format that can be stored in a buffer or file, or be transmitted over a network connection. Data can be classified in two types: simple data types and composite data types. A simple data type cannot be decomposed into other types. Examples of simple data types are integers, floats and chars. Composite data types on the other hand are composed of other data types. Classes, structs and arrays are examples of composite data types. In ROOT I/O data serialization is made possible through a special class method, called a Streamer. A Streamer finds in memory the data members of the object it was invoked from and "streams" them to a buffer, from where they can be written to file. There are two ways in which a Streamer of an object (i.e. object D) can invoke Streamers from other objects:

1. If object D holds data members that are objects the user wants to write to file (i.e. objects C and E), the Streamer method of C and E are invoked automatically. This could cause a chain-reaction of Streamers being invoked if C and E hold other objects of their own.

2. If class of which object D is instantiated from, has one or more parent classes, the Streamers of these parent classes are invoked automatically. This causes a chain-reaction of Streamer invocation throughout the inheritance tree of object D.

Figure 1 illustrates the two chain-reaction invocations of a Streamer method. Class D is derived from class B and holds object data members of class C. Whenever the Streamer of class D object is invoked, the Streamer of the derived class B object is also invoked. Likewise all of the Streamers of the class C objects are invoked. Since B does not derive from another

class and C does not hold objects from other classes or is derived from other classes the streaming process for an object of D is completed.

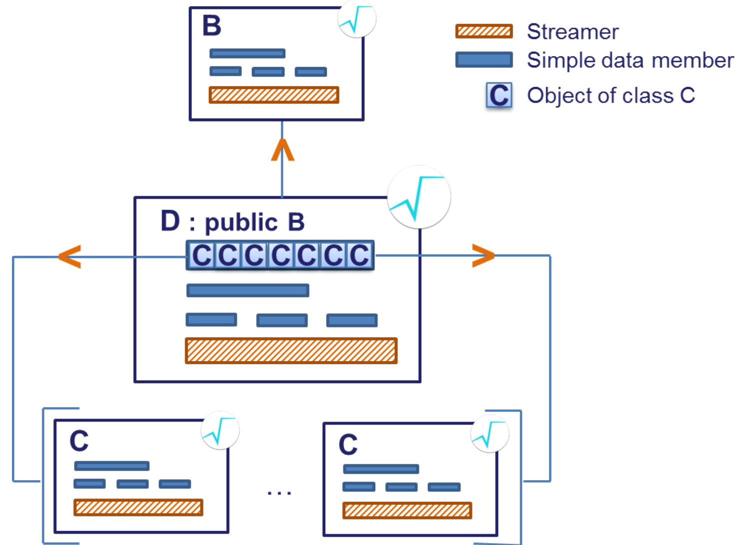

Figure 1: Invoking Streamers of classes an object inherits or holds.

There are two remarks on the Streamer method that need to be kept in mind when working with it. The first is the misunderstanding that if an object X holds an object Y that holds an object X, this would cause a never-ending loop of Streamer invocations. This is not correct, because at any given point in a run-time of a program, the number of objects that are instantiated (i.e. have memory allocations) are finite. The invocation chain would stop when one of the objects has not yet allocated memory for the other object, and therefore cannot invoke its Streamer. The second remark is that each object, even if referenced multiple times (by pointer or reference), are only written once. Further references to the same objects result in only a reference (i.e. its offset in the streamer buffer with respect to its first occurrence) to this object to be written to the buffer.

A Streamer can either be automatically generated by ROOT or can be implemented by the user. Generating Streamers is discussed in more detail in Section 3.1.3.

3.1.2 ClassDef Macro

ROOT provides Run-Time Type Information (RTTI) for any object the user wants to write to file. RTTI is a form of type introspection, which essentially means that an object can expose information about itself, such as what data type it is or whether it is derived from a certain class. ROOT does so in the form of a macro, called ClassDef. This macro is added at the end of the header file of a class and declares several methods; some of which are implemented in-line, and some that need an implementation to be generated. Generating implementations for ClassDef methods is explained in Section 3.1.3.

Using the ClassDef macro is optional when one wants to use ROOT I/O to write objects to file. The advantage however is that it can increase the I/O speed, because it provides the Streamer with the properties of the object. Since ClassDef only declares public static methods in a class, the memory layout of a class instance is in no way altered.

There are several version of the ClassDef macro that should be used for specific situations:

- **ClassDef:** the standard macro that should be used if the two ClassDef versions below do no apply for a given situation. This macro assumes that the class has a virtual function table and will add its own virtual methods to it. This implies that the class' destructor should be declared virtual.
- **ClassDefOverride:** this macro version should be used in two specific situations. The first one being when a class includes header files of other classes that have already a ClassDef macro defined. The ClassDef methods from the included headers should be overridden with the ClassDef methods of the class of interest with this macro. The second situation is when the class uses the `override` keyword from C++11/14.
- **ClassDefNV:** should be used whenever the class does not have virtual methods and no other classes derive from this class.

Listing 3.1: Usage of a ClassDef macro in MyClass.h

```
#include <Rtypes.h>
class MyClass {
 public:
  ...
 private:
  ...
 ClassDef(MyClass, 1);
};
```

In Listing 3.1 one can see an example of how the ClassDef macro should be included in the header file of a dummy class "MyClass". Also note the inclusion of the ROOT header file "Rtypes.h", which declares all the versions of ClassDef. ClassDef takes as arguments the name of the class and a class version number. Version numbering is used for schema evolution in ROOT, which keeps track of changes in the class layout. Each time a class changes its data layout the class version number should be incremented by the user. Schema evolution ensures that data produced by a program at one point in time can be worked with at a later point in time even though the classes that make up the program (slightly) changed. Example of changes are number format changes (i.e. float to double), additional data members, removal of data members. The user is responsible for handling data conversion and assignments of newly added data members when reading back an older version for example.

3.1.3 Dictionaries

A dictionary is what connects ROOT I/O with the classes that make up an application. It holds descriptive information about all the classes that wish to benefit from ROOT I/O. It contains information such as a list of all global variable and functions, a list of all classes and a full description of the data members and methods of each class. Moreover, it facilitates the implementation of automatically generated Streamer methods, in order to serialize data members of an object into a buffer.

A dictionary for one or more classes can be generated by **rootcling** - an interactive C++ interpreter. It takes as input the header files of the classes and generates a dictionary as output. See Chapter 4 for its usage.

3.1.4 LinkDef File

In a LinkDef header file one can specify the contents of a dictionary that should be generated by `rootcling`. Inside a LinkDef file the user lists pragma statements that link classes to the generated dictionary. For each class the user wants to integrate ROOT I/O in, the class name should be listed in one of the pragma statements, like shown in Listing 3.2.

Listing 3.2: Pragma to link class to dictionary

```
#pragma link C++ class MyClass+;
```

Notice the trailing "+" sign at the end of the pragma. It enables rootcling to use the new I/O system, introduced after ROOT 3, making it possible to fully support STL collections and increases runtime performance. It should have been the default, but in some cases it break code [3], so the user needs to add it manually.

A LinkDef file always start with the contents of Listing 3.3. This is to tell `rootcling` to not make a dictionary of everything (which is the default), but rather the user-defined entries.

Listing 3.3: LinkDef standard pragmas

```
#ifdef __ROOTCLING__
// do not link everything (turn link "off")
#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

// include classes within classes
#pragma link C++ nestedclasses;
#endif
```

In cases of listing class templates one should pay special attention to the order of their entries. If a certain class X holds a specialized instance of a class template Y, then in the LinkDef file the pragma statement of Y should be listed before the entry of X. More on pragma statements in LinkDef files can be found in Section 17.5.1 of [4].

3.2 Data Persistence

ROOT I/O provides the necessary functionality to stream data from memory to a buffer. For applications that generate considerate amount of data (i.e. much more than the capacity of the memory) it is not sufficient to simply keep the serialized data in buffer. It is more convenient to write these buffers to a file on disk, so data can be retrieved and loaded back into memory at a later point in time. Writing to file and reading from file is also implemented in ROOT I/O.

[3] `https://root.cern.ch/selecting-dictionary-entries-linkdefh`

3.2.1 TFile Object and ROOT File

In order to achieve maximum compressing, ROOT stores data in machine-independent binary format files, called ROOT files (i.e. extension is .root). ROOT files go hand-in-hand with dictionaries, as the dictionaries can translate back and forth between binary data and OOP data types. In fact, at the time of writing to a ROOT file, dictionary information is stored within a a ROOT file as `TStreamerInfo` objects, which describe the detailed the class layout of all objects stored in the file. If an object of classes A, B and C are stored, there will be three `TStreamerInfo` objects stored as well; one for each class A, B and C. By locating and reading these TStreamerInfo objects one know the layout of all objects written in the file. There are 3rd party readers (a Javascript one, ROOTIO.js), totally independent of the ROOT code, that can read ROOT files and, reconstruct the objects. A TFile object is used to access ROOT files in a programming environment. Creating a new TFile object and ROOT file is as simple as:

Listing 3.4: Creating a TFile

```
TFile *f = new TFile("myfile.root", "NEW");
f->Close();
```

In the first line of Listing 3.4 a new ROOT file (`myfile.root`) is created and can be accessed by the instantiated TFile object (`f`). The option `NEW` creates a new file and, only if the file did not already exists, opens it. The `RECREATE` option would overwrite an already existing file. In the second line the TFile is closed and the result is the creation of an empty ROOT file. For any operation that would be done between these two lines, the `myfile.root` would serve as the storage facility on disk. A TFile can contain objects and directories. Figure 2 illustrates the memory layout of a ROOT file containing several objects.

Figure 2: Structural memory layout of a ROOT file.

The file header gives information about the ROOT file, such as the version of ROOT that was used to create this file. `TKey` is a class that acts as a header for an object in a ROOT file. It contains information about the corresponding object, such as the size of the object (compressed as well as uncompressed), its class name, and creation date and time. Compression is simply achieved by compressing each buffer (typically using zlib or lzma) before it is written to the file. See section 11.1 of [4] for more details on the information in a ROOT file header and TKey.

3.2.2 WriteObject() and GetObject()

In order to write one or more object into a ROOT file, calling the corresponding Streamer methods is not enough. These will merely serialize the objects and place them in a temporary buffer in memory. ROOT provides a method to automatically write the contents of a buffer to file, named `WriteObject()`. Besides the pointer to the object, `WriteObject()` takes as an argument the unique name of the TKey object that will be created for the object that needs to be streamed. Listing 3.5 shows an example of its usage.

Listing 3.5: Writing an object to file

```
TFile *f = new TFile("myfile.root", "NEW");
MyClass *obj_w = new Myclass();
f->WriteObject(obj_w, "obj_name");
f->Close();
```

At the end of Listing 3.5 `myfile.root` will contain an instance of MyClass in serialized format with its corresponding `TKey` object named `obj_name`. The pipeline that an object X in memory goes through in order to be serialized into a ROOT file is described as following:

1. A `TKey` object is created in the opened `TFile`
2. A `TBuffer` object is created inside the `TKey` object
3. Streamer of X is invoked and its `TStreamerObject` is created
4. `TBuffer` object is filled according to `TStreamerObject`
5. Buffer is written to ROOT file
6. `TBuffer` object is freed from `TKey` object

Reading back one or more objects from a ROOT file and loading them into memory requires the `GetObject()` method, which serves as the counterpart of `WriteObject()`. It takes as arguments the key name that was used to write the object to file and a newly constructed object of the same class. Listing 3.6 shows how one could retrieve the object that was stored to ROOT file in Listing 3.5.

Listing 3.6: Reading an object from file

```
TFile *f = TFile::Open("myfile.root");
MyClass *obj_r = new Myclass();
f->GetObject("obj_name". obj_r);
f->Close();
```

4 Making a Class Persistent

This chapter serves as a step-by-step tutorial on making C++ classes persistent using ROOT I/O. The steps below will focus on a single class called "MyClass". It shall be assumed that ROOT is installed and configured correctly (see Appendix A for instructions on how to install ROOT).

STEP 1

Insert the ClassDef macro at the end of the header file of the class, MyClass.h. See Section 3.1.2 for choosing the right macro version. Also include the header file "Rtypes.h" that declares the macro.

Listing 4.1: Insert ClassDef macro

```
#include <Rtypes.h>
class MyClass {
 public:
  ...
 private:
  ...
 ClassDef(MyClass, 1);
};
```

STEP 2

Create an I/O constructor or a public constructor that has zero parameters or one or more parameters that are set to default values. This constructor is needed so that during reading the basic object can be created.

Listing 4.2: ROOT I/O constructor

```
class MyClass {
 public:
  MyClass(TRootIOCtor*) { }  // I/O constructor
  ...
};
```

STEP 3

Initialize uninitialized pointers to composite data types in the I/O constructor.

Listing 4.3: Initialize pointers to composite data types

```
class MyClass {
 public:
  MyClass(TRootIOCtor*) {
    bar_ = nullptr;     // do not allocate space here
  }
```

```
    ...
  private:
    Foo foo_ = nullptr;    // already initialized
    Bar bar_;              // initialize in I/O ctor
};
```

STEP 4

Add the class name to the LinkDef header file.

Listing 4.4: LinkDef.h

```
#ifdef __ROOTCLING__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ nestedclasses;
#pragma link C++ nestedtypedef;

#pragma link C++ class MyClass+;  // do not forget trailing +

#endif
```

STEP 5

Create dictionary from the header file and LinkDef file and compile to object file.

Listing 4.5: rootcling

```
$ rootcling MyClassDict.cxx -c my_class.h LinkDef.h
$ g++ -c -fpic ‘root-config --cflags‘ DictMyClass.cxx
```

STEP 6

Compile the implementation file and link previously created dictionary to executable or shared library.

Listing 4.6: Compile and link executable

```
$ g++ -c -fpic ‘root-config --cflags‘ my_class.cc

// shared library
$ g++ -shared -o ‘root-config --glibs‘ libMyClass.so MyClassDict.o my_class.o

// executable
$ g++ -o ‘root-config --glibs‘ MyClass MyClassDict.o my_class.o
```

A Makefile implementation of step 5 and 6 can be found in Appendix B.

5 Workflow

This chapter describes the general workflow of working with BioDynaMo and ROOT throughout this project. It includes Github version control, ROOT I/O integration in BioDynaMo classes, testing for bugs and debugging.

5.1 ROOT I/O in BioDynaMo

The simulation architecture of BioDynaMo is comprised of the following four abstraction layers, which originate from the Cx3D simulation program [2]:

1. **Cell:** describes the biological processes that influence a whole neuron. The user defines modules that govern the behavior of a neuron entity.
2. **Local Biology:** describes the local biological properties within a neuron. The user defines modules that impose rules on the sphere and cylinders that respectively represent the soma and dendrites of a neuron.
3. **Physics:** describes the physical processes that are exerted on cellular substances, such as diffusion. The user has limited interaction with this layer; methods can be called that influence the mechanics of a neuron (i.e. growth, branching).
4. **Spatial Organization:** describes how elements are organized in space. There is no user interaction with this layer.

Each of these layers are composed of several classes. One specific class called `ECM`, which stands for extracellular matrix, contains a list of all the objects that are active during a simulation, which are instances of the classes that make up these four layers. Therefore there is only a single instance of `ECM` throughout the simulation; a singleton class design pattern. This is very convenient in terms of making the simulation data persistent with ROOT I/O. Once all the classes of BioDynaMo are persistent, simply invoking the Streamer method of the `ECM` object would cause a chain-reaction of Streamer invocations on all the runnable objects in the simulation (see Section 3.1.1).

The BioDynaMo persistence project began with a `git clone` from the BioDynaMo repository and the compilation thereof.

Listing 5.1: First BioDynaMo build

```
$ git clone https://github.com/BioDynaMo/biodynamo.git
$ cd biodynamo

$ mkdir build && cd build
$ cmake ..
$ make
```

Subsequently, a branch named `ahmad` [4] was created on the BioDynaMo repository to facilitate the code integration of ROOT I/O. On this branch the general workflow is illustrated in the flowchart display in Figure 3.

[4] `https://github.com/BioDynaMo/biodynamo/tree/ahmad`

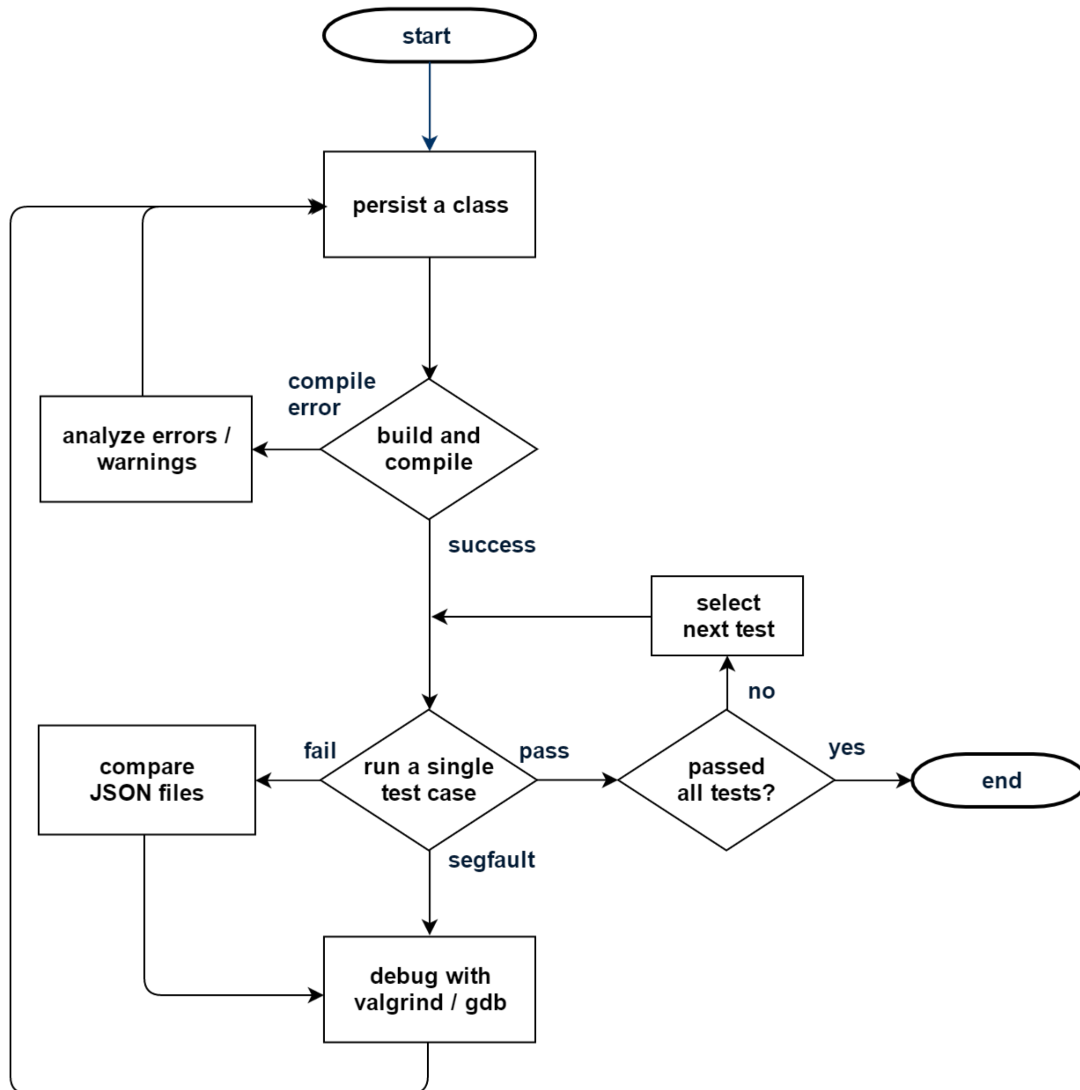

Figure 3: Workflow chart of integrating ROOT I/O in BioDynaMo

5.2 Persisting and Building

Making a BioDynaMo class persistent was done as described in Chapter 4. Not all classes were considered to be made persistent, because some classes served as interfaces for the user to create his or her own modules from. As mentioned earlier, the `ECM` singleton object contains all active objects in the simulation. Therefore this was the object that would be used for the `WriteObject()` and `GetObject()` methods to actually persist all the data members to file. Section 5.3 shows how this is actually implemented.

The difference between the method described in Chapter 4 and the method used for persisting BioDynaMo classes is steps 5 and 6. BioDynaMo uses CMake to generate Makefiles in order to build the simulation executable. ROOT provides CMake users a macro that generates the `rootcling` compile command, called `ROOT_GENERATE_DICTIONARY`. For every BioDynaMo class that was made persistent, its header file was added to the list of header files that were

used by this macro to generate a single dictionary. It's usage is described in Listing 5.2.

Listing 5.2: Generating a dictionary with CMake

```
ROOT_GENERATE_DICTIONARY(MyClassDict my_class.h LINKDEF LinkDef.h)
```

Detailed instructions on how to integrate ROOT I/O into a CMake project can be found on the ROOT How-To website [5]. After a class was presumably persisted, the BioDynaMo executable would be rebuild (`cmake ..`) and recompiled (`make`) in the build directory of BioDynaMo.

5.3 Testing

Once the executable was built and compiled, it was possible to run simulation test cases. BioDynaMo comes with a total of twelve automated test cases that can be used to validate the correctness of changing code or writing new code. At the end of a simulation the values of certain data members of the active objects are parsed and written to JSON files. For each test case there is a 'reference JSON file' that was obtained by the original Cx3D simulation. Comparing the JSON file generated from a BioDynaMo simulation with its corresponding reference file will reveal whether or not the code generates the expected outcome. If it does not, it means that there is a bug introduced in the code. Listing 5.3 shows in pseudocode how this was done *before* ROOT I/O was implemented.

Listing 5.3: Testing prior to ROOT I/O implementation

```
simulate()          // simulate according to specified test
ECM::writeToJson()  // write all data members to JSON format
Compare()           // compare the generated JSON with reference JSON
if (not_equal)      // if above case is not equal
 JSON_to_file()     // write JSON to file
```

In order to examine the effect ROOT I/O has on the generated data, the ECM object should be written to file, read back from file and an additional JSON file should be made. If the JSON file is equal to the reference JSON file it would mean the test had passed and data members were correctly persisted. Listing 5.4 shows in pseudocode the order of events that was needed to correctly test ROOT I/O implementation.

Listing 5.4: Testing after ROOT I/O implementation

```
simulate()                // simulate according to specified test
ECM::writeToJson()        // write generated data members to JSON format

WriteObject(ECM, "ecm")   // write ECM object to file under name "ecm"
ecm_r = GetObject("ecm")  // read back object "ecm" and point to it with ecm_r
ecm_r::writeToJson()      // write data members to JSON format

Compare()                 // compare the generated data with reference data
CompareRoot()             // compare ROOT retrieved data with reference data
if(not_equal)             // if either of above cases is not equal
 JSON_to_file()           // write both JSON to file
```

[5] https://root.cern.ch/how/integrate-root-my-project-cmake

All the test cases are derived from a base class called `BaseSimulationTest`. It was in this class that the original comparison between simulation generated data and reference data was done. It was therefore natural to implement the writing and reading to and from a ROOT file in `BaseSimulationTest`, as well as implementing the additional comparison. The JSON file generated from the ECM object read back from the ROOT file will be further referred to as the "ROOT JSON file".

5.4 Debugging

A test failure is useful in pointing out that there is a bug in the code that was changed or added. From a comparison between the ROOT JSON file and reference JSON file one can find indications of which data members are influenced. However finding the cause for the bug was often not trivial.

GNU `gdb` is useful in setting breakpoints in the code and tracking down the bug in order to limit the lines of code one needs to review. The general workflow for gdb with the BioDynaMo executable would be as shown in Listing 5.5.

Listing 5.5: Debugging BioDynaMo with gdb

```
$ gdb runBiodynamoTests                  // load debugging symbols
$ run -- --gtest_filter=<testName>*      // run a single test
$ break source.cc:<line#> <condition>    // set breakpoints
$ p/<option> <variable>                  // print out local variable
$ where                                  // print out stack trace
```

See a gdb cheat sheet for a full list of gdb commands and options [6].

In some cases the test would not even complete, because of a segmentation violation. These kind of problems can best be debugged with `valgrind`.

Listing 5.6: Debugging BioDynaMo with valgrind

```
$ valgrind --track-origins=yes --log-file="valgrind_log" --suppressions=
    $ROOTSYS/etc/valgrind-root.supp ./runBiodynamoTests -- --gtest_filter=<
    testname>*
```

5.5 Profiling

Persisting the simulation state will undoubtedly introduce overhead. Table 1 shows the time it took for each test case to write the entire simulation state a single time to a ROOT file. These numbers are obtained by simply computing the difference in timestamps before and after the `WriteObject(ECM)` call.

The write times are of most interest to the user, since writing to file is done periodically throughout the runtime of the program, but reading is only done when necessary. The average write time of 1.1 seconds is negligible when writing to file is done each half hour for example. Also the file size is much less in comparison to writing the data members in

[6] http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Table 1: Time spent for a single write of `ECM` to ROOT file for each BioDynaMo test case and the corresponding file size

Test name	Write time (ms)	File Size (kB)
DividingCellTest	1175	6
DividingModuleTest	1000	45
IntraCellularDiffusionTest	1175	895
Figure5Test	1050	11
Figure9Test	1350	1600
MembraneContactTest	1000	8
NeuriteChemoAttractionTest	1050	164
RandomBranchingModuleTest	1050	24
SimpleSynapseTest	1050	8
SmallNetworkTest	1175	692
Average	**1108**	**345**

JSON format, where the size ranges from 58 kB to 23.5 MB for the same test cases. These JSON files did not even include all of the data members. Moreover, the lowest level of compression was used for the generated ROOT files in Table 1. It can be concluded that ROOT is capable of persisting a simulation state quickly and compactly.

5.6 Current Status and Future Work

Currently there are two issues that make BioDynaMo not perfectly persistent.

1. **idCounter issue:** several classes hold a static counter variable called `idCounter`. ROOT I/O does not write static data members to ROOT file, because they do not belong to any specific class instance. Therefore some test cases fail due to a difference in `idCounter` values between generated JSON files and reference JSON files.
2. **Shared_ptr issue:** `shared_ptr`'s are not yet supported by ROOT I/O. Support is expected in October 2016.

For future work, there are two main features of ROOT I/O that should be explored for the BioDynaMo platform.

1. **Partial I/O:** Currently the entire simulation state is being written and read from file, while some applications written for the BioDynaMo platform might not utilize all its data members. ROOT can provide a more efficient solution by putting data members into branches of a `TTree` object. The branches of a TTree can build separate and independent buffers of the data members they hold. This makes it possible to cherry-pick branches that should be persisted for a certain BioDynaMo application. Not only does this reduce disk space, but also enhances access times.
2. **Network distribution:** Since the plan for BioDynaMo is to be run in parallel on a hybrid cloud-computing infrastructure, it will be necessary to have a distributed simulation state for each node in the network. This requires ROOT files to be accesses over a network, which is a functionality provide by ROOT through `TNetFile` objects.

Whether or not ROOT will be a feasible solution (in comparison to other fault tolerant distributed file systems like Hadoop Distributed File System) needs to be investigated.

6 Troubleshooting

During this project several issues and errors were encountered that prevented data members from becoming persistent. This chapter lists all of the encountered issues and errors and their corresponding solutions. These solutions might not be valid in general, but might give an indication to its cause. References to specific BioDynaMo classes in errors are replaced with "MyClass".

PROBLEM 1

Data members of class templates are not picked up as persistent data members, even though the class itself is made persistent.

SOLUTION: Class templates need separate entries in the LinkDef header file with the types that are used in the code.

Listing 6.1: Class template in LinkDef.h

```
#pragma link C++ class MyClass<Foo>+;
```

Not having this entry in LinkDef.h would result in a data member `MyClass<Foo> foo_`, for example, not being streamed to file. Having data members of the type `MyClass<Bar>` would require an additional entry in the LinkDef header file. Also make sure that the class template pragma entry is mentioned before any class pragma entry that holds an instance of it.

PROBLEM 2

Data members of type `std::vector<unique_ptr<T>>` and `std::array` are not streamed to file, even though their corresponding classes are persistent and other data members are streamed.

SOLUTION: Since 22-08-2016 ROOT supports the I/O of these C++11 data types. Update to a version of ROOT greater than or equal to 6.07/07. Encountering an unsupported data type would require the user to handle its I/O manually by defining the memory layout for rootcling of that data type. For example, Listing 6.2 shows how an `unique_ptr` data member would be streamed with a ROOT version that does not support its I/O.

Listing 6.2: ifdef trick

```
#ifdef __ROOTCLING__
  std::vector<MyClass*> foo_;
#else
  std::vector<unique_ptr<MyClass>> foo_;
#endif
```

PROBLEM 3

Occasionally test cases would not complete because of segmentation violations such as in Listing 6.3.

Listing 6.3: Segfault when running a test

```
*** Break *** segmentation violation

===========================================================
There was a crash.
This is the entire stack trace of all threads:
===========================================================
#0  0x00007f355fb6ea0c in __libc_waitpid (pid=5757, stat_loc=stat_loc
entry=0x7fffc2a53740, options=options
entry=0) at ../sysdeps/unix/sysv/linux/waitpid.c:31
#1  0x00007f355faf4232 in do_system (line=<optimised out>) at ../sysdeps/
    posix/system.c:148
#2  0x00007f3560dab013 in TUnixSystem::StackTrace() () from /home/ahmad/
    Desktop/root/lib/libCore.so
#3  0x00007f3560dada1c in TUnixSystem::DispatchSignals(ESignals) () from /
    home/ahmad/Desktop/root/lib/libCore.so
#4  <signal handler called>
#5  0x00007f35613fa59c in ROOT::
    delete_bdmcLcLspatial_organizationcLcLSpatialOrganizationNodelE-
    bdmcLcLphysicscLcLPhysicalNodegR (p=0x7f35610b4a90 <vtable for TString
        +16>) at /home/ahmad/Desktop/biodynamo/build/bdmDict.cxx:2604
#6  0x00007f3560d6f333 in TClass::Destructor(void*, bool) () from /home/ahmad
    /Desktop/root/lib/libCore.so
...
```

Running valgrind on this specific test case would result in mentions of conditional jumps depending on uninitialized value(s), as shown in Listing 6.4.

Listing 6.4: Valgrind of test case segfault

```
Conditional jump or move depends on uninitialised value(s)
    at 0x4F112B3: ROOT::
        delete_bdmcLcLspatial_organizationcLcLSpatialOrganizationNodelE-
        bdmcLcLphysicscLcLPhysicalNodegR(void*) (bdmDict.cxx:2954)
    by 0x5622332: TClass::Destructor(void*, bool) (in /home/ahmad/Desktop/
        root/lib/libCore.so)
    ...
Uninitialised value was created by a heap allocation
    at 0x4C2B0E0: operator new(unsigned long) (in /usr/lib/valgrind/
        vgpreload_memcheck-amd64-linux.so)
    by 0x4F22AB1: ROOT::new_bdmcLcLsimulationcLcLECM(void*) (bdmDict.cxx
        :3440)
    by 0x56213F8: TClass::New(TClass::ENewType, bool) const (in /home/ahmad/
        Desktop/root/lib/libCore.so)
    ...
```

From this stack trace it is apparent that whenever a new bdm::simulation::ECM object would be instantiated by ROOT I/O, there were value(s) uninitialized.

SOLUTION #1: In ECM.h there is a data member called initial_node_ of type SpatialOrganization<PhysicalNode>. This member would not be initialized when root made a

SpatialOrganization<PhysicalNode> object. This was only done in the implementation. So initializing initial_node_ to nullptr in the TRootIOCtor resulted in no more segfaults.

SOLUTION #2: In other cases solution #1 might have not resolved the issue. Another cause for a segfault could have been not including a class name in the LinkDef header file. In BioDynaMo some test cases have their own classes defined that interact with BioDynaMo interfaces. These classes however should also be persisted according to Chapter 4.

References

[1] L. J. Breitwieser, R. Bauer, M. Manca, and F. Rademakers, "Porting a Java-based Brain Simulation Software to C++," Sep. 2015. [Online]. Available: https://doi.org/10.5281/zenodo.46842

[2] F. Zubler and R. Douglas, "A framework for modeling the growth and development of neurons and networks," *Frontiers in Computational Neuroscience*, vol. 3, p. 25, 2009. [Online]. Available: http://journal.frontiersin.org/article/10.3389/neuro.10.025.2009

[3] F. Zubler, A. Hauri, S. Pfister, R. Bauer, J. C. Anderson, A. M. Whatley, and R. J. Douglas, "Simulating cortical development as a self constructing process: a novel multi-scale approach combining molecular and physical aspects," *PLoS Comput Biol*, vol. 9, no. 8, p. e1003173, 2013.

[4] R. Brun and F. Rademakers, "ROOT - An Object Oriented Data Analysis Framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1, pp. 81–86, 1997.

Appendix A

Instructions on installing and configuring ROOT on a personal UNIX system.

STEP 1

Clone ROOT source files from the Github repository to a local drive. To retrieve another (possible more stable) release of ROOT, use the `git checkout` command with a tag specifying the version.

Listing 6.5: Get ROOT source files

```
$ git clone https://github.com/root-mirror/root.git

// (optional) retrieve specific release (i.e. 6.07/07)
$ cd root
$ git checkout -b v6-07-07 v6-07-07
```

STEP 2

Configure the ROOT installation with the `configure` script provided in the source files.

Listing 6.6: Configure ROOT installation

```
// cd to ROOT top directory
$ cd root
$ ./configure --all
```

Most likely there will be some packages missing that the ROOT build depends on; these need to be installed manually. The easiest way to find out which package is missing and needs to be installed is by iteratively running the `./configure --all` command and installing the package that will be complained about. Listing 6.7 is an example on how to this can be done. On `https://root.cern.ch/build-prerequisites` the names of the relevant packages are listed.

Listing 6.7: Find and install dependencies

```
$ ./configure --all
configure: dpkg-architecture MUST be installed

$ sudo apt-get install dpkg-dev
```

STEP 3

Build ROOT using GNU `make`. Set the number of cores for `make` to the highest number the system allows in order to speed up the build process.

Listing 6.8: Build ROOT

```
// replace "2" with number of preferred cores
$ make -j2
```

STEP 4

Set environment variables for every terminal by running the following commands. `<root_dir>` is the directory where the root directory is placed.

Listing 6.9: Set environmental variables

```
$ gedit ~/.bashrc

// add the following line at the end of this file
// replace <root_dir> with actual directory
source <root_dir>/bin/thisroot.sh
```

Open a new terminal and run the command `root` in it. Congratulations, ROOT has been successfully installed.

```
 ------------------------------------------------------------------
| Welcome to ROOT 6.07/07                     http://root.cern.ch |
|                                   (c) 1995-2016, The ROOT Team |
| Built for linuxx8664gcc                                         |
| From heads/master@v6-07-06-1365-gee6992f, Aug 22 2016, 17:26:11 |
| Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'      |
 ------------------------------------------------------------------
```

Appendix B

Listing 6.10: Makefile for persistent executable

```
ROOTCFLAGS    = ‘root-config --cflags‘
ROOTGLIBS     = ‘root-config --glibs‘

CXX           = g++
CXXFLAGS      = -I$(ROOTSYS)/include -O -Wall -fPIC -Wno-reorder
LD            = g++
LDFLAGS       = -g
SOFLAGS       = -shared

TARGET        = my_class
DICTNAME      = myClassDict
SOURCES       = my_class.cc
HEADERS       = my_class.h
LINKDEF       = $(wildcard *LinkDef.h *Linkdef.h)

CXXFLAGS     += $(ROOTCFLAGS)
GLIBS         = $(ROOTGLIBS)

all: $(TARGET)
clean:
  @rm *.o *.pcm $(TARGET) $(DICTNAME).cc *.root 2>/dev/null || true

$(DICTNAME).cc: $(HEADERS) $(LINKDEF)
  @rootcling -f $@ -c $(CXXFLAGS) $(HEADERS) $(LINKDEF)
$(TARGET).o: MyClassDict.cc
  @$(CXX) -c $(DICTNAME).cc $(CXXFLAGS)

$(TARGET): $(SOURCES) $(DICTNAME).o
  @$(CXX) -o $(TARGET) $(SOURCES) $(CXXFLAGS) $(GLIBS) $(DICTNAME).o
```

In order to customize this Makefile to build an executable for a specific project, the following steps need to be executed:

1. All the source files of the project should be added to `SOURCES`.
2. All the header files that belong to classes that have ROOT I/O integrated in them should be added to `HEADERS`.
3. Change the target name (`my_class`), dictionary name (`myClassDict`) to the desired names.
4. Make sure there is only one LinkDef header file in the directory. Its name must end with "LinkDef" or "Linkdef" (i.e. `myClassLinkdef.h` is correct, but `LinkDefMyClass.h` is not). Simply `LinkDef.h` is fine as well.

Listing 6.11: Make commands

```
// Build and run executable
make && ./my_class

// Clean built files
make clean
```