Coconut: Typestates for C++

Arwa Hameed Alsubhi
University of Glasgow
arwaalsubhi99@gmail.com

Ornela Dardha
University of Glasgow
ornela.dardha@glasgow.ac.uk

This paper introduces typestates in C++ by creating a typestates library, as a tool, which provides templates for protocol definitions, and then statically check that objects of the class are behaving accordingly. This project is expected to aid programmers by helping them detect violations of protocols that they themselves define, making the programming process more flexible, practical, and trouble-free. The tool is available in this Github repository [1].

1 Introduction

Typestate, as a concept, has been applied to many object-oriented languages; it is a protocol specification to verify that operations on objects are performed in order. Recently, new languages were mainly created to integrate *typestates*; such as Fugue, Plaid and Obsidian [11, 7, 10], while others extended existing languages with *typestate* tools like Mungo in Java, Papaya in Scala, Typestates Pattern in Rust and more [5, 14, 15, 12, 9, 8]. Although C++ is one the most commonly used programming languages, unfortunately, there is only one attempt to implement *typestate*, as an expressive code for a finite state machine [2]. However, it handles the checking for violations by using an intermediate wrapper class, which is impractical; therefore, this project aims to smoothly do the checking process without an intermediate function or class to provide simplicity and a better user interface.

2 Methods

The approach of this project is to build a static library to be a *typestate* tool in C++. The library uses features and techniques such as Templates and Namespaces which are provided by C++ language [18, 16]. This is explained in detail in the tool's demo video [6] and this Tutorial website [4].

2.1 Protocol templates

We will go over the implementation with a known example File reading process. if we have a file class with three methods Open(), Read(), and Close(), the Open() method should be called before other methods. So, the protocol would be written in a form of a state machine with the use of the *enumeration* for defining all possible states. As for transitions between states, they would be defined by using templates in the library. Using templates Meta-programming concept in C++ allows the reshaping of different data structures, and it has many other features [17]. The templates are extracted as in listing 1.

```
1 using TypestateTool::map_transition;
2 using TypestateTool::map_protocol;
```

Listing 1: Extracting templates

Submitted to:
13th Workshop on Programming Language Approaches to Concurrency-Communication-cEntric Software.

After extracting the templates, the transitions between states are written as in listing 2. By using these templates, the compiler will generate copies of the protocol templates implementations and save them at compile-time. Then, these generated copies would be analyzed and used in the checker Class.

```
using File_protocol=  map_protocol<map_transition<FileState::CLOSED, FileState::
    OPEN, &File::Open>,
map_transition<FileState::OPEN, FileState::READ_MODE, &File::Read>,
map_transition<FileState::READ_MODE, FileState::READ_MODE, &File::Read>,
map_transition<FileState::READ_MODE, FileState::CLOSED, &File::Close>
>;
```

Listing 2: Protocol Definition

2.2 Protocol checker

A Checker class in the library is built to do *typestates* checking. However, to perform the checking, the defined protocol first is linked to the class via defining a Maroc Directive function [3] and it is called in the library `Assign_to_Class()` as displayed in listing 3.

```
Assign_to_Class(File, File_protocol);
```

Listing 3: Linking a protocol to class

Later, to verify that the implementation of a File object in listing 4 is according to the protocol, the Checker class uses a function called `Check_transition()` which takes the object implementation and the defined protocol (`File_protocol`) as parameters. However, the `Check_transition()` function has a pointer which is a technique that points to the function's address in the main memory [13].

```
int main() { File file = File();
    file.Open(); file.Read(); file.Read();file.Close();return 0;}
```

Listing 4: Object implementation

This pointer firstly points to the first state that is defined by the user in the first define transition. In the File class example, the pointer would initially point to a CLOSED state. Then it moves between states according to the object implementation. However, every time before the pointer moves, the `Check_transition()` function checks if it was a valid transition or not. For example, file.Open() moves from the CLOSED state to an OPEN state, which is a valid transition, so the pointer would move to the OPEN state as it is explained in Appendix 4.1. However, if we start with file.Read() or File.Close(), the pointer would not move and the `Check_transition()` function would not let the program compile see Appendix 4.2.

3 Results, Discussion and Future Work

To conclude, the Coconut tool is a typestates tool which was developed for the C++ language. However, some of examples are run to test the effectiveness of the Coconut tool and to compare its features with other *typestates* tools. In terms of effectiveness, these examples showed that the tool was in fact a useful typestate tool capable of checking violations for defined protocols in most cases. Nevertheless, there are few cases where some bugs exist; however, they could mainly because of the inconsistent movements of the pointer in the Checker class, and they would be tackled in the future work. Regarding the features of programming languages, the Coconut tool supports some features that other tools like Mungo and Papaya support, such as branching and recursion. However, some features are not yet part of the tool but they would be included in the future, such as using fields to store objects that have a protocol.

References

[1] Coconut tool. https://github.com/ArwaAlsubhiM/CPP_Typestate.

[2] Fluent c++. URL: https://www.fluentcpp.com//.

[3] Preprocessor macros (c/c++). URL: https://docs.microsoft.com/en-us/cpp/preprocessor/macros-c-cpp?view=msvc-170.

[4] Tutorial website for typestates in c++. https://arwaalsubhi99.wixsite.com/typestateuesrtesting/.

[5] The typestate pattern in rust - cliffle. URL: http://cliffle.com/blog/rust-typestate/.

[6] Typestates in c++. https://youtu.be/UqdHB17yRtQhttps://youtu.be/UqdHB17yRtQ.

[7] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, et al. Plaid: a permission-based programming language. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 183–184, 2011. doi: http://dx.doi.org/10.1145/2048147.2048197.

[8] Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, 2009. doi: http://dx.doi.org/10.1007/978-3-642-03013-0_10.

[9] Eric Bodden and Laurie Hendren. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer*, 14(3):307–326, 2012. doi: http://dx.doi.org/10.1007/s10009-010-0183-5.

[10] Michael Coblenz. Obsidian: a safer blockchain programming language. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 97–99. IEEE, 2017. doi: http://dx.doi.org/10.1109/ICSE-C.2017.150.

[11] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004. doi: http://dx.doi.org/10.1007/978-3-540-24851-4_21.

[12] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 177–190, 2006. doi: https://dl.acm.org/doi/abs/10.1145/1217935.1217953.

[13] Lars Haendel. The function pointer tutorials. *newty. de [online]*, pages 1–13, 2005. URL: http://www. newty. de/fpt/functor. html.

[14] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. Papaya: Global typestate analysis of aliased objects. In *23rd International Symposium on Principles and Practice of Declarative Programming*, pages 1–13, 2021. doi: http://dx.doi.org/10.1145/3479394.3479414.

[15] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Science of Computer Programming*, 155:52–75, 2018. doi: http://dx.doi.org/10.1016/j.scico.2017.10.006.

[16] Ray Lischner. *C++ In a Nutshell: A Desktop Quick Reference*. " O'Reilly Media, Inc.", 2003. URL: https://books.google.co.uk/books?hl=en&lr=&id=Q4iP1mkfdtsC&oi=fnd&pg=PT7&dq=C%2B%2B+In+a+Nutshell:+A&ots=w0k7cAPU2s&sig=bQCaw42M0TG2esaegy7uxvfzqZU&redir_esc=y#v=onepage&q=C%2B%2B%20In%20a%20Nutshell%3A%20A&f=false.

[17] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos. Debugging c++ template metaprograms. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 255–264, 2006. doi: https://dl.acm.org/doi/abs/10.1145/1173706.1173746.

[18] David Vandevoorde and Nicolai M Josuttis. *C++ Templates: The Complete Guide, Portable Documents*. Addison-Wesley Professional, 2002. URL: https://books.google.co.uk/books/about/C++_Templates.html?id=yQU-NlmQb_UC&redir_esc=y.

4 Appendices

4.1 First Appendix

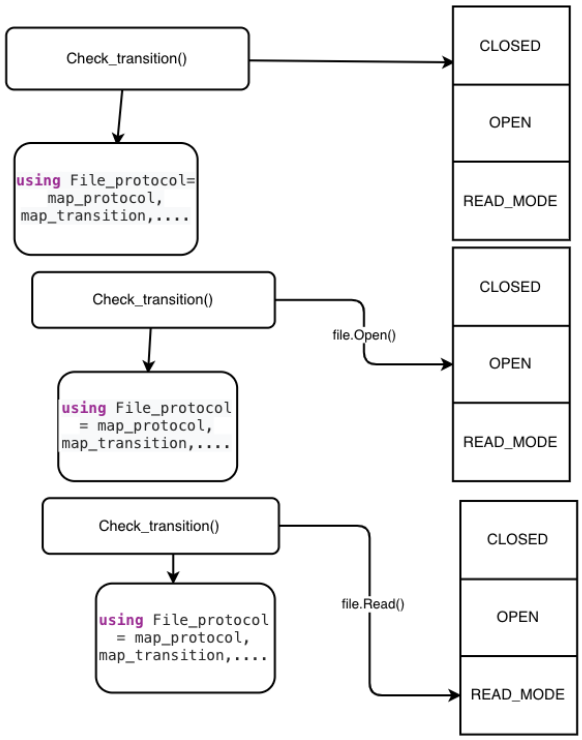

Figure 1: `Check_transition()` function approach

4.2 Second Appendix

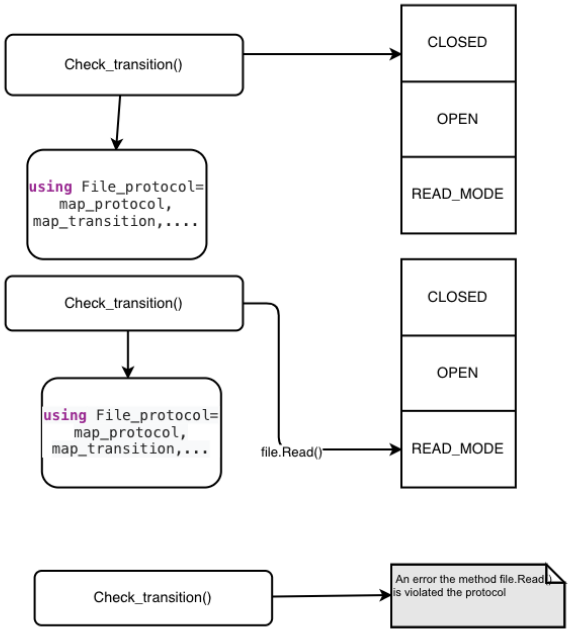

Figure 2: `Check_transition()` function produce Error