Coconut: Typestates for Embedded Systems

Arwa Hameed Alsubhi(✉) and Ornela Dardha

University of Glasgow, Glasgow, UK
a.alsubhi.1@research.gla.ac.uk, ornela.dardha@glasgow.ac.uk

Abstract. Typestate programming defines object states and actions to improve software safety by ensuring operations on objects follow the correct sequence. While its adoption in object-oriented languages has increased, limitations persist in the features supported. Typestates are particularly useful in embedded systems for operation sequencing, yet examples in this area are scarce. We introduce Coconut, a C++ tool that leverages typestate programming with templates for specifying typestates and combining static type checking and dynamic analysis to ensure proper class instance behaviour. It uniquely supports advanced programming features like branching, recursion, aliasing, concurrency, and optional typestate visualisation, facilitating idiomatic object-oriented programming with inheritance. Illustrating its effectiveness, we apply Coconut to actual embedded system projects, advancing the field by introducing a comprehensive set of features and practical examples for implementing typestate programming.

Keywords: Typestate · C++ · Embedded Systems

1 Introduction

Software development in critical sectors such as finance and healthcare necessitates thorough testing and maintenance to protect sensitive data and ensure human safety. A key aspect of this maintenance involves the use of protocols that dictate the sequence of operations for objects and help reduce errors, as highlighted by [49]. For example, in message-passing processes, establishing a connection is a prerequisite before any message transmissions can occur. While some protocols are straightforward, others in real-world systems are complex and challenging, requiring ongoing research to refine enforcement approaches.

Typestate analysis is one such approach that could effectively enforce specific sequences of operations on objects, thus managing and reinforcing protocols, as discussed by [54]. To illustrate further, consider the `LightSwitch` class shown in Fig. 1. The protocol dictates that the `LightSwitch` must not be turned on if it is already in the **on** state, nor turned off if it is in the **off** state. If the object `ls` in Fig. 2 attempts to call `SwitchOn()` twice without an intervening `SwitchOff()`, typestate analysis statically detects and prevents this protocol violation. For more details on how this analysis is enforced in Coconut, refer to Sect. 2.

© IFIP International Federation for Information Processing 2024
Published by Springer Nature Switzerland AG 2024
I. Castellani and F. Tiezzi (Eds.): COORDINATION 2024, LNCS 14676, pp. 219–238, 2024.
https://doi.org/10.1007/978-3-031-62697-5_12

```
class LightSwitch{
public:
void SwitchOn(){
std::cout<<"Switching On\n";
}
void SwitchOff(){
std::cout<<"Switching Off\n";
}
};
```

Fig. 1. LightSwitch Class

```
int main() {
LightSwitch ls;
(ls->*&LightSwitch::SwitchOn)();
(ls->*&LightSwitch::SwitchOff)();
return 0;
}
```

Fig. 2. LightSwitch Client Code

Leveraging the advantages of typestate, several new programming languages, such as Vault, Fugue, Plaid, and Obsidian, have been developed with a focus on integrating typestate as a key feature, as detailed in [18–20,51]. Additionally, tools like Mungo, Java Typestate Checker (JaTyC), and Papaya have been applied to object-oriented programming languages, such as Java or Scala, to facilitate typestate checking [5,10,32,34,41]. Although these projects leverage typestate checking to support various programming features, they still lack some critical capabilities. For example, most of these projects do not support inheritance, with the notable exception of JaTyC, which fully supports it [5]. Furthermore, many typestate-based projects either eliminate aliasing, such as Mungo [34], or impose limitations on it, like Plaid [3].

These limitations arise from several factors. For instance, aliasing can complicate the management of object typestates. This is because changes made by one alias can affect the state of the object as seen by another alias, leading to inconsistencies and undermining the accuracy of the object's current state analysis. Similarly, implementing typestate in programs that use inheritance is challenging because subclasses may introduce new methods and behaviours that do not align with the parent class's typestate. This can lead to a situation where the subclass's unique behaviours need separate enforcement mechanisms, adding complexity to the system design. To address these complexities, many typestate initiatives selectively focus on features that align with their core objectives, thus limiting their scope to ensure practical implementation that accommodates diverse needs. This project aims to provide such support by focusing on essential features that enhance the application of typestates across various scenarios.

In this paper, we examine the application of typestate analysis in embedded systems, such as medical devices, which require precise sequences of operations for reliable functionality. Given this requirement, typestate tools become valuable for addressing these operational needs in such systems. However, despite the growing interest in formal methods for verifying and validating embedded systems, such as [12,13,29–31], there is a noticeable lack of research on typestates in these systems. Formal methods such as Extended Finite State Machine (EFSM) [2] and Event-B [1] use states and events for system modelling. EFSM offers detailed behaviour modelling at a lower level, while Event-B supports multi-level system modelling. However, their complexity and resource demands

pose challenges for systems with limited resources and constrained update processes. Typestate as mentioned in [40], offers a practical approach to behaviour representation and the ability to enhance code modularity. This modularity can improve the readability and maintainability of the codebase, thus simplifying coding and updating workflows [35].

This paper presents the Coconut tool, a C++ library with typestates. We selected C++ for this implementation due to its prominence in embedded systems and proven effectiveness in high-performance applications, as evidenced by [12,26,38,52]. C++20's brief inclusion of contract attributes-later removed-highlighted a gap in verifying object behaviour [16,46]. Typestates could address this, yet integrating them into C++ is challenging due to static reflection limitations and C++'s complex library creation process [11,55]. The existing ProtocolEncoder (ProtEnc) library offers typestate analysis but is limited by performance issues and restricted feature support, as highlighted in [53]. Our tool overcomes these limitations by applying distinct approaches in checking and analysing and offers more features beyond ProtEnc.

Contributions. The key contributions of this paper are as follows:

- **Typestate Analysis:** Coconut provides typestate templates for protocol definition and supports these with a comprehensive analysis to ensure class instances adhere to their protocols. Detects violations at compile-time and incorporates run-time assertions for enhanced robustness (Sect. 2).
- **Embedded Systems Case Studies:** We showcase the application of Coconut through case studies, such as the LightSwitch and Pillbox. The Pillbox case study is grounded in the actual deployment of the Pillbox system in a healthcare setting [12] (Sects. 2 and 3).
- **Comprehensive Programming Feature Set:** Coconut integrates a broad range of programming features, including branching, recursion, aliasing, concurrency, inheritance, and typestate visualisation. Coconut is the first tool to integrate these features comprehensively (Sect. 3).
- **Evaluation Study:** We conducted a comparative evaluation to benchmark Coconut's performance against three distinct case studies, each implemented in C++ and evaluated across different metrics (see Sect. 4). Furthermore, we tested each Coconut example to ensure its effectiveness in identifying bugs and enforcing typestate.

2 The Coconut Tool

This section explores the integration, system overview, template mechanics, and static and dynamic analysis capabilities of the Coconut tool.

2.1 Integration, Compatibility, and Usage

Coconut is a C++ library tailored for C++20 and later versions, offering specialised functionalities. It utilises CMake [33], a well-known cross-platform build

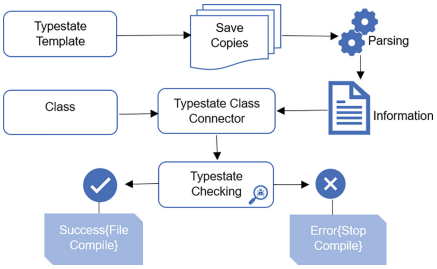

Fig. 3. Coconut Typestate Checking Process

system, to simplify the compilation process across various operating systems and environments. This integration facilitates compatibility with various external dependencies, such as the Boost library [15]. Coconut offers flexible integration into C++ projects, allowing developers to selectively include its header file based on their needs, thus enabling or bypassing its features accordingly. Comprehensive instructions for its installation and usage are available in its repository[1]. The library implements a type-safe state machine using template metaprogramming [43], Boost.Hana [14] and functional programming principles. It provides mechanisms to define states, transitions, and rules for a typestate, along with utilities for tracking instances and visualising the state machine's structure. This approach is particularly useful for compile-time checks and enforcing correct state transitions in a type-safe manner.

2.2 System Overview

Upon integrating the Coconut library, developers can specify behavioural protocols for their class objects using the provided templates. The system parses and analyses these templates to collect information about all possible states and permitted behaviours. As explained in Fig. 3, This data is passed to the `TypestateClassConnector`, which establishes a linkage between each class and its corresponding typestate, ensuring compliance with defined typestate guidelines throughout the codebase. The `State_Manager` class is responsible for typestate verification. It manages this by wrapping Pointers to Member Functions (PMFs) that are used to call class functions. These PMFs are accessed using operators like `.*` and `->*`. The `State_Manager` acts as a gatekeeper, employing a `constexpr` template function to check whether an object is in the correct state before it allows a function call to proceed. It verifies against the typestate rules stored in a tuple. If the function call aligns with the rules, the `State_Manager` updates the object's state in the tuple to reflect the new state. If a function call violates the rules, the checking process will detect this and stop the compilation of the program.

[1] https://zenodo.org/records/10853974.

```
BETTER_ENUM(LightSwitchState, int, OFF, ON);
using LightSwitchProtocol = Typestate_Template<
State<+LightSwitchState::OFF,&LightSwitch::SwitchOn,+LightSwitchState::ON>,
State<+LightSwitchState::ON,&LightSwitch::SwitchOff,+LightSwitchState::OFF>>;
```

Fig. 4. LightSwitch Typestate

2.3 Templates and State Management in Coconut

Coconut utilises the concept of *enumeration* by leveraging either built-in *enumeration* or the `BETTER_ENUM` library. The `BETTER_ENUM` library [6] enhances `enum` functionality by enabling conversions to and from strings, which aids in typestate visualisation. Moreover, it offers compile-time validation capabilities not available with built-in `enums`, often utilised in examples within our tool. Using templates, the typestate specification is defined as a Finite State Machine (FSM). This FSM is modelled through a template-based `struct` and implemented via template meta-programming [43], which allows for flexibility in accommodating various data types. This versatility enables the management of diverse classes, such as File and Sound, within the Coconut tool. States and transitions are expressed as follows:

- **Typestate_Template:** This template serves as the container for the FSM, utilising *variadic templates* [28] to accommodate an arbitrary number of state transitions.
- **State:** Each state transition is modelled as follows:
  - **Current_State:** Indicates the state the object is currently in.
  - **Pointer_To_Allowable_Function:** Defines the member function pointer that is allowed to transit the object from the Current_State to the Next_State.
  - **Next_State:** Specifies the state to which the object transitions after the function is called.
- **Wrapper Templates for State Transition Management**: Each class is connected with the `Typestate_Template` and wrapped by a manager that embeds typestate logic directly into the class structure. This wrapper generates code that uses the tuple stored in `Typestate_Template` to check typestate configurations within the Coconut framework. Upon instantiation, each class begins in a state defined by the `current_state` in the `Typestate_Template`. This ensures that the object begins its life-cycle from the correct state. The process of state transition checking is initiated whenever a state-changing method is called. If the transition is valid according to the `Typestate_Template`, the method is called, moving the object to its next state. If a transition is invalid, static assertions prevent compilation, while runtime assertions manage function calls in dynamic behaviours.

To demonstrate how templates function within the Coconut tool, consider the LightSwitch example mentioned in Sect. 1, as illustrated in Fig. 1. Initially,

the typestates of the LightSwitch are defined in the `Typestate_Template`. This includes specifying permissible states and transitions between them as seen in Fig. 4. Following the specification, a wrapper transforms the LightSwitch class to incorporate these typestates. This setup integrates the logic necessary for state transitions directly into the structure of the class. In Fig. 2, upon instantiation, the `ls` is set to the OFF state, as defined by the first state entry in the **Typestate_Template**. When the `SwitchOn()` method is called, Coconut checks the typestate configurations stored in the tuple to verify if the transition to the ON state is permissible using `SwitchOn()`. If the transition is valid, the `ls` moves to the ON state and the program compiles successfully. If the `ls` in Fig. 2 in line 4 calls `SwitchOn()` the compilation will stop, enforced by static assertions.

2.4 Static and Dynamic Analysis in Coconut

In the Coconut tool, the code analysis starts with static analysis during the compilation phase. This involves checking that the code behaves correctly when typestate-related functions are called within `if-constexpr` conditions or loops. Static analysis ensures that these calls correctly manage state transitions based on the compile-time conditions and the defined rules, preventing potential logic errors before the program runs. Additionally, static analysis includes aliasing analysis, which tracks how multiple references to the same object might impact the object's state during these function calls when executed within a monolithic context. It also examines inheritance structures to ensure that derived classes adhere to the state management rules of their base classes.

After successful compilation, dynamic analysis takes over during code execution to manage conditions not addressed during the static phase. This includes handling `if-else` branches that respond to runtime conditions involving changes in internal state or effects of predefined variables. For example, a vending machine may enter a maintenance state triggered by internal diagnostics, which is checked at runtime. The dynamic aliasing part of the analysis examines how aliases affect an object's state through function calls in dynamic segments of the program, using a compositional approach. This analysis also supports concurrent method calls across multiple threads, ensuring alignment with type states.

Limitations and Future Work. Coconut's current limitations include a lack of support for dynamic data structures like arrays and linked lists, restricting its application in complex data manipulation. It also lacks full support for direct interactions with users or real-time data from external systems, such as processing live data feeds from sensors or streaming data from online services. Additionally, advanced concurrency challenges such as synchronisation, race conditions, and deadlocks. These areas are targeted for enhancement in future updates.

3 PillBox Case Study

This section demonstrates how our tool can be applied to a real-world scenario using the smart PillBox, an embedded medical system designed to ensure patients take their medications on time, as detailed in [12].

```
class PillBox {
public:
    PillBox() {
      DrawersBox = new std::vector<Drawer*>();
    }
    void Activate_PillBox() {
       std::cout<<"PillBox is active now!\n";
    }
    void addDrawers(Drawer* d) {
      DrawersBox->push_back(d);
    }
    Drawer* Process_System_Time(int h, int m) {
      for (Drawer* d : *DrawersBox) {
        if (h == d->get_the_hour() && m == d->get_minutes())
          { return d; }}
      return nullptr;
    }
    void Deactivate_PillBox() {
      std::cout<<"PillBox is Deactivated now!\n";
    }
    void Switch_ON(Drawer* d) {
      redled.setRedLed("ON");
      std::cout<<"It's time to take "<<d->get_pill_name()<<"!\n";
    }
    void Switch_OFF(Drawer* d) {
      redled.setRedLed("OFF");
      std::cout<<"Drawer with the pill "<< d->get_pill_name()<<" is
          closed\n";
    }
    void Blink(Drawer* d) {
      redled.setRedLed("Blinking");
      std::cout<<"Please close the drawer "<< d->get_pill_name() << "\n";
    }
private:
    std::vector<Drawer*>* DrawersBox;
    RedLed redled;
};
```

Fig. 5. PillBox Class

3.1 PillBox Original vs Coconut Version

Remark 1. Before discussing the implementation of PillBox in Coconut, it is important to review the original deployment as described in [12] and the identified issues. PillBox is a programmable device that allows for the addition or removal of modular drawers and was originally modelled using the Asmeta framework [24], a model-based engineering platform which uses four hierarchical Abstract State Machines (ASM) to define system behaviours. Since ASM models cannot be executed directly on the hardware, translating these models into C++ code with the tool Asm2C++ [23] was crucial. The translated C++ code

is used for debugging and validating system performance, which is unachievable with ASM alone. This translated code has significant hardcoding and extensive conditional statements, reflecting the complexities of the original ASM code and making it challenging to trace execution and reason about the code's behaviour. Furthermore, a new ASM model is required each time a drawer is added to the system, which currently only supports up to three drawers, leading to inefficiencies. Comparing the Coconut tool implementation with this setup is essential to understand how each manages the system. A small fragment of the code from the original implementation can be found in Appendix A.

Typestate Solution: In response to challenges identified in the original PillBox implementation [12], we propose a solution based on typestates facilitated by the Coconut tool. Typestates offer a structured framework for modelling PillBox behaviour, effectively addressing these challenges. By utilising typestates within Coconut, we establish a unified framework for the PillBox, regardless of the number of drawers. Unlike the original approach, which required separate modelling and validation for each level of the hierarchical structure using hardcoding, typestate specifications are defined once for the entire PillBox. This process simplifies the checking and verification of client code, reducing the complexity and redundancy associated with modelling and validating multiple scenarios compared to the original implementation using ASM, thus making it easy to trace execution and reason about the code's behaviour.

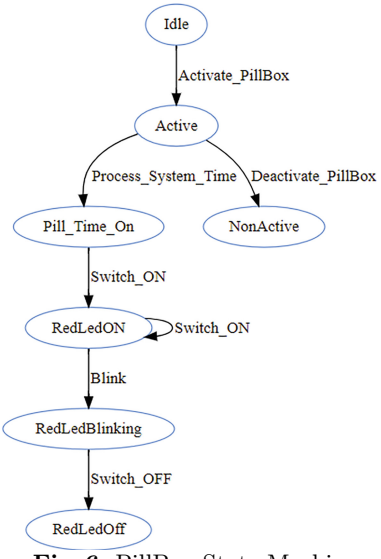

Fig. 6. PillBox State Machine

Now, we will demonstrate how Coconut implements typestates for the smart PillBox and see how typestate checking ensures the system functions correctly as illustrated in the state machine (Fig. 6). In the PillBox class (Fig. 5), multiple drawers hold pills with specific intake times, and a RedLed, initially off. Each drawer is represented as a class (Fig. 7), detailing the pill and its intake time. Similarly, the RedLed in the `PillBox` class is also a class (Fig. 8).

PillBox Typestate Description. Each `PillBox` instance starts in the (`Idle`) state. Upon activation, the `RedLed` switches on to indicate the time to take a pill from a specific `Drawer`. The `RedLed` remains illuminated until the patient opens the `Drawer` and retrieves the pill. After pill retrieval, the `RedLed` blinks to close the `Drawer`, then switches off. This sequence repeats for each `Drawer` when its corresponding pill time arrives. This typestate is expressed using `State` and

`Typestate_Template`, depicted in Fig. 9, and represented as an FSM in Fig. 6. To demonstrate, in Fig. 10, the `pillbox1` instance of the `PillBox` class is set to the `Idle` state when is created in line 12, as defined by the first state entry in the **Typestate_Template**. When the `Activate_PillBox()` method is called on `pillbox1`, Coconut checks the typestate configurations stored in the tuple to verify if the transition to the `Activated` state is permissible using `Activate_PillBox()`. If the transition is valid, `pillbox1` moves to the `Activate` state, and the program compiles successfully. However, if `pillbox1` attempts to call `Activate_PillBox()` again after line 16, the compilation will stop, enforced by static assertions. Note that methods outside the state machine, like `addDrawers`, are considered "anytime" methods and are not subject to typestate checks.

```
1  class Drawer {
2  public:
3      Drawer(std::string
4      pillName, int hour, int
           minute):
5      pillName_(pillName),
6      hour_(hour), minute_(minute) {}
7      std::string get_pill_name()
8      {return this->pillName_;}
9      int get_the_hour()
10     {return this->hour_;}
11     int get_minutes()
12     {return this->minute_;}
13 private:
14     std::string pillName_;
15     int hour_, minute_;
16 };
```

Fig. 7. Drawer Class

```
1  class RedLed {
2  public:
3      void setRedLed(std::string
           state)
4      {this->RedLedState = state;}
5  private:
6      std::string RedLedState;
7  };
```

Fig. 8. RedLed Class

3.2 Discussion of Programming Features

This section outlines Coconut's key features, including branching, recursion, aliasing, concurrency, and inheritance. These features play roles in decision-making, task simplification, maintaining consistent object states, supporting multi-threaded interactions, and enhancing code scalability. They are useful for developing maintainable software for complex systems like the smart PillBox, as illustrated by [21], which provides insights into various scenarios where such features are utilised in software development for complex systems.

Branching and Recursion. Coconut supports branching in programming, enabling objects to follow multiple paths within a program. For instance, as demonstrated in Fig. 10, when the `pillbox1` instance is in the (Active)

```
using PillBox_typestate = Typestate_Template<
State<+domain::Idle,&PillBox::Activate_PillBox,+domain::Active>,
State<+domain::Active,&PillBox::Process_System_Time,+domain::Pill_Time_On>,
State<+domain::Pill_Time_On,&PillBox::Switch_ON,+domain::RedLedON>,
State<+domain::RedLedON,&PillBox::Switch_ON,+domain::RedLedON>,
State<+domain::RedLedON,&PillBox::Blink,+domain::RedLedBlinking>,
State<+domain::RedLedBlinking,&PillBox::Switch_OFF,+domain::RedLedOFF>,
State<+domain::Active,&PillBox::Deactivate_PillBox,+domain::NonActive>>;
```

Fig. 9. PillBox Typestate Specification

```
1  static void OperateRedLed(PillBox& p, Drawer* d){
2      for(int i=0;i<5;i++){
3         (p->*&PillBox::Switch_ON)(d);
4       }
5      (p->*&PillBox::Blink)(d);
6      (p->*&PillBox::Switch_OFF)(d);
7  }
8  int main() {
9      Drawer* drawer1 = new Drawer("Panadol", 3, 50);
10     Drawer* drawer2 = new Drawer("Piriton Antihistamine", 8, 40);
11     constexpr bool EnableDrawersOperations = true;
12     PillBox pillbox1;
13     PillBox& ptr_pillbox1 = pillbox1;
14     pillbox1.addDrawers(drawer1);
15     pillbox1.addDrawers(drawer2);
16     (pillbox1->*&PillBox::Activate_PillBox)();
17     if constexpr (EnableDrawersOperations) {
18        Drawer* d = (pillbox1->*&PillBox::Process_System_Time)(3,50);
19        OperateRedLed(ptr_pillbox1, d);
20     }
21     else {
22       (pillbox1->*&PillBox::Deactivate_PillBox)();
23     }
24     return 0;
25 }
```

Fig. 10. PillBox Client Code

state, it can selectively choose which method to invoke. It can proceed with `Process_System_Time()` as shown in line 18, transitioning to the `(Pill_Time_On)` state, or it can call `Deactivate_PillBox()` to transition to the (NonActive) state. Coconut also supports loops, as illustrated in Fig. 10, where the `for` loop in line 2 continuously turns on a red LED with `Switch_ON()` and signals when to take a pill, repeating this process until the incrementer reaches 5.

Unrestricted Aliasing. Coconut analyses relationships between different variables and objects in the program. For instance, it analyses the relationships

between `pillbox1`, `ptr_pillbox1` as seen in line 13 in Fig. 10, and any other variables that may reference the same `pillbox1` and aliases are directly linked to their respective instances and states. When `pillbox1` or `ptr_pillbox1` in lines 16 or 19 calls a function, Coconut checks this data against the predefined typestate. If it conforms, Coconut updates the instance's state, and this change is then reflected across all aliases.

Concurrency. In C++, concurrency and threading are managed using the `std::thread` class, which facilitates the execution of functions, function objects, or lambda expressions. Wrappers as discussed in Sect. 2, encapsulate typestate on objects. For concurrency, these wrappers use hidden `mutexes` to enforce exclusive execution of certain code blocks by one thread at a time, with runtime checks ensuring adherence to typestate for consistent state transitions. Consider the scenario in Fig. 10, where the main function executes two threads with distinct operations, as detailed in Fig. 11. Here, Coconut integrates `mutexes` within wrappers, employing runtime checks to enforce adherence to specific typestate rules, thus preventing state inconsistencies. For instance, if thread `t1` initiates system activation, it acquires a mutex, blocking thread `t2` from starting until `t1` completes and releases the mutex. Conversely, if `t2` starts first, encountering a non-activated system due to `t1` not having run yet, `t2's` execution halts.

Inheritance. Coconut enforces typestate rules across superclasses and subclasses in line with the Liskov Substitution Principle (LSP) [36], ensuring that subclasses can seamlessly replace superclasses without impacting program functionality. It analyses four inheritance scenarios to ensure methods and behaviours

```
1 std::thread t1([&pillbox1]{
2     (pillbox1->*&PillBox::Activate_PillBox)();
3 });
4 std::thread t2([&pillbox1]{
5      Drawer* d = (pillbox1->*&PillBox::Process_System_Time)(3,50);
6 });
```

Fig. 11. Concurrency Example

conform to typestate requirements. Particularly in scenarios where both classes have typestates, Coconut ensures that subclasses adhere to the specifications of both their own class and their superclasses, as elaborated in the repository.

3.3 Coconut vs. State-of-the-Art Typestate-Based Tools

Coconut retains all previously discussed features and introduces an optional typestate visualisation feature. This feature helps developers represent typestates diagrammatically, improving clarity and understanding. Developers can use the `Visualise_TypestateTemplate<enum>` function which uses Graphviz [27] to visualise protocols and create diagrams, such as the one shown in Fig. 6. Before concluding this section, we summarise our programming feature contributions and compare them with other typestate-based tools, as shown in Table 1. Compared to other typestate-based projects, *Coconut manages to provide support for all discussed features*, unlike other tools that lack other features.

Table 1. Coconut vs typestate-based Projects Features. A checkmark in the table indicates the presence of a feature. Most of these features are implemented at compile time. However, features like Unrestricted Aliasing and Concurrency are offered at runtime, except for Concurrency in JaTyC21, which is offered at compile time.

	Branching	Recursion	Inheritance	Safe Aliasing	Unrestricted Aliasing	Concurrency	Visual
Mungo	✓	✓					
JaTyC21	✓	✓		✓		✓	
JaTyC22	✓	✓	✓				
DSI_Rust		✓					✓
Plaid	✓	✓		✓		✓	
Clara	✓	✓					
Plural	✓	✓		✓			
Papaya	✓	✓		✓	✓		
Accumulation	✓	✓		✓			
Fugue	✓	✓	✓	✓			
Obsidian	✓	✓		✓			
ProtEnc		✓		✓			
Coconut	✓	✓	✓	✓	✓	✓	✓

4 Evaluation Study

Benchmarks and Metrics. In our evaluation study, we aim to assess the performance of the Coconut tool, which applies typestate analysis, by comparing it with three distinct case studies. Initially, we evaluate the performance using a straightforward example of a LightSwitch embedded system as a baseline [8]. Next, we examine the PillBox embedded system, previously discussed in Sect. 3,

which involves hardcoding and utilises an Abstract State Machine [12]. Finally, we compare Coconut's performance against an existing typestate-based tool, ProtEnc, implemented in C++ [53]. We have selected four key performance metrics for this analysis, which are:

- **Compilation Time (CT)**: Measures the time required to compile code.
- **Run Time (RT)**: Measures the duration of program execution.
- **Memory Usage (MU)**: Assesses the amount of execution memory usage.
- **Code Complexity**: Evaluates the complexity of the source code.

These metrics are selected to assess the software's efficiency, reliability, and maintainability, aspects that are particularly important in embedded system environments. Compilation and execution times reflect speed and efficiency, memory usage helps gauge resource optimisation, and code complexity provides insight into maintainability. These selected measures align with standard practices in embedded systems [42,45,48].

Process. We used a Python script with libraries like pandas, lizard, openpyxl, time, psutil, and Libclang17dev [17,25,37,39,44,56] to measure metrics such as compile time, runtime, and memory usage. Each script ran 100 times to find average results. For code complexity, we checked the number of tokens, lines of code (NLOC), and Cyclomatic complexity. *Notably, the original PillBox system heavily relied on runtime hardcoding and checks, using the Arduino library* [4] *to monitor data from an embedded device like system time and the RedLed status. To ensure a fair comparison with Coconut, which mainly checks at compile time without direct device access, we adjusted the PillBox to move all checks to compile time.* For data collection, we tested the modified system's compile time over 100 iterations, each processing unique parameters such as pill names, times (in hours and minutes), and the red LED status, mimicking real-world operations. We applied the same testing approach to the Coconut PillBox version to align with our goal of evolving Coconut into a typestate monitoring tool.

Results, Conclusion and Future Improvement After experimentation, as depicted in Fig. 12, we observed that Coconut incurs a higher compile-time overhead compared to other benchmarks. However, it exhibits slightly better performance in terms of runtime and memory usage. In assessing code complexity, Coconut achieves lower values for Cyclomatic complexity, NLOC, and token count than the other benchmarks. These lower complexity metrics suggest that code written with Coconut might be simpler and less complex, which can make it easier to understand and maintain. *Note that the results for Pillbox in the table refer to the modified version.* Additionally, the effectiveness of Coconut in identifying bugs was assessed through case studies, with detailed findings available

Metrics	LightSwitch Original	LightSwitch Coconut	PillBox Original	PillBox Coconut	ProtEnc Original	ProtEnc Coconut
Compile Time (Milliseconds)	850 ▼	2889 ▲	2200 ▼	3339 ▲	2713 ▼	3387 ▲
Run Time (Milliseconds)	27 ▲	13 ▼	18 ▲	12 ▼	11 ▲	9 ▼
Memory Usage (Kilobyte)	1778 ▲	1774 ▼	1805 ▲	1784 ▼	1818 ▲	1776 ▼
NLOC	25 ▲	24 ▼	201 ▲	105 ▼	84 ▲	48 ▼
Tokens	145 ▲	116 ▼	1355 ▲	743 ▼	461 ▲	309 ▼
Cyclomatic Complexity	3 ▲	2 ▼	21 ▲	19 ▼	6 ▲	5 ▼

Fig. 12. Metrics Results

in the Coconut repository. We plan to broaden our evaluation in the future to include more comprehensive case studies in embedded systems and to incorporate additional metrics and expand testing to include more complex scenarios.

5 Conclusion, Related and Future Work

Related Work. Typestate was introduced by Strom and Yemini [50] to enhance program reliability through compile-time semantic checks. This foundational concept focused on ensuring correct variable declaration and initialisation, setting the stage for future developments in typestate programming.

In the development of typestate in programming languages, projects like Vault [19], Fugue [20], and Plaid [3,51] played pivotal roles. These projects expanded typestate into programming languages or language extensions, allowing the definition of resource protocols or class states. This approach mirrors the strategies seen in Rust's Typestate Pattern Tool [47] and the ProtocolEncoder (ProtEnc) in C++ [53], which prioritise augmenting existing languages. Furthermore, the concept of session types, as explored in tools like Mungo, JaTyC, and Papaya [5,32,34], emphasises separate object protocol definitions and static class instance checking against these protocols.

Aliasing, where multiple references can be assigned to a single object, is treated differently across various typestate-oriented projects. Projects like Vault [19], Fugue [20], Plaid [3,51], and Plural [9] have developed unique constructs and modes to manage aliasing. Vault uses adoption and focus, creating linear and non-linear references to control access to objects. Fugue distinguishes

between NotAliased and MayBeAliased modes, directly addressing the aliasing issue. Plaid and Plural introduce access permissions, blending aliasing and access control for the management of object references.

Supported Programming Features. Branching and recursion are seamlessly integrated into tools like Mungo [34] and Papaya [32], using language-specific statements for recursion management. Concurrency is uniquely tackled in typestate tools. Plaid [3] prioritises access permissions, while JaTyC21 [41] focuses on tracing state changes for safe concurrent computations. Inheritance, crucial in object-oriented programming, is addressed by JaTyC22 [5] and Fugue [20], allowing subclasses to extend superclasses' typestates.

Embedded Systems. Formal methods are essential in the verification and validation of embedded systems in healthcare and robotics. Bonfanti et al.'s systematic literature review highlighted this necessity in medical software development, ensuring human safety [13]. Such methods include the use of UML-B state machines and class diagrams to model and analyse the HD Machine for treating kidney failure [29], evaluating various states and transitions essential to its operation. In robotics, finite state machines have been demonstrated as effective approaches for controlling robot behaviour and performance [7,22]. For autonomous vehicles, a model integrating finite-state machine and reinforcement learning methods is highlighted, crucial in navigating cut-in situations [31]. Another model employs FSM to address the parking challenges faced by autonomous vehicles, utilising sensors and system requirements [30].

Conclusion and Future Work. To conclude, in this paper, we introduced the Coconut tool and discussed the usage of templates to define typestates for objects in C++. Our tool ensures the conformity of the typestate definitions on objects by conducting a thorough analysis and covering a full spectrum of programming features to date. We presented the architecture of the tool and showcased its support for different programming features through case studies in the embedded systems industry, to introduce and emphasise the importance of typestate as an approach for validating and verifying such systems. We assessed the Coconut tool through many tests for Coconut case studies, also we conducted an evaluation study with three benchmarks using four metrics. In future work, we aim to enhance Coconut by addressing its performance limitations. Our vision is to transform Coconut into a typestate monitoring tool capable of processing embedded device data, facilitating evaluation and comparison with actual embedded systems.

Acknowledgements. Supported by the UK EPSRC New Investigator Award grant EP/X027309/1 "Uni-pi: safety, adaptability and resilience in distributed ecosystems, by construction". Additionally, this work has partial support from the Royal Embassy of Saudi Arabia Cultural Bureau. We thank Simon Gay for his valuable comments on the paper.

A PillBox Original Implementation

See Figs. 13, 14.

```
void pillbox_FULL::r_Main_seq() {
    for (auto _compartment : Compartment::elems)
        if (true) {
            if ((drugIndex[0][_compartment]
                < (unsigned int) ((time_consumption[0][_compartment]).size()))) {
                { //par
                if (!opened[0][_compartment] & openSwitch[_compartment]) {
                    opened[1][_compartment] = true;
                }
                if (opened[0][_compartment] & !openSwitch[_compartment]) {
                    opened[1][_compartment] = false;
                }
                if ((redLed[0][_compartment] == OFF)) {
                    if ((time_consumption[0][_compartment][drugIndex[0][_compartment]]
                        < systemTime)) {
                        r_pillToBeTaken(_compartment);
                    }
                }
                if ((redLed[0][_compartment] == ON)
                    & !((systemTime - compartmentTimer[0][_compartment]) >= 1)
                    & opened[0][_compartment] & !openSwitch[_compartment]) {
                    r_pillTaken_compartmentOpened(_compartment);
                }
                if ((redLed[0][_compartment] == ON) & !opened[0][_compartment]
                    & openSwitch[_compartment]) {
                    r_compartmentOpened(_compartment);
                }
                [...]
```

Fig. 13. Code Snippet of PillBox from the original implementation in [12]

```
    for (const auto& _compartment : Compartment::elems) {
        time_consumption[0].insert( { _compartment, [&]() { /*<--- caseTerm*/
            if(_compartment==compartment1)
            return std::vector<unsigned int> {1, 20, 30};
            else if(_compartment==compartment2)
            return std::vector<unsigned int> {40, 50, 60};
            else if(_compartment==compartment3)
            return std::vector<unsigned int> {9, 20, 30};
        }() });
        time_consumption[1].insert( { _compartment, [&]() { /*<--- caseTerm*/
            if(_compartment==compartment1)
            return std::vector<unsigned int> {1, 20, 30};
            else if(_compartment==compartment2)
            return std::vector<unsigned int> {40, 50, 60};
            else if(_compartment==compartment3)
            return std::vector<unsigned int> {9, 20, 30};
        }() });
    }
    systemTime = 0;
    for (const auto& _compartment : Compartment::elems) {
        name[0].insert( { _compartment, [&]() { /*<--- caseTerm*/
            if(_compartment==compartment1)
            return "Tylenol";
            else if(_compartment==compartment2)
            return "Aspirine";
            else if(_compartment==compartment3)
            return "Moment";
        }() });
        name[1].insert( { _compartment, [&]() { /*<--- caseTerm*/
```

Fig. 14. Code Snippet of PillBox 2 from the original implementation in [12]

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge University Press, Cambridge (2010). https://doi.org/10.1017/CBO9781139195881
2. Alagar, V.S., Periyasamy, K.: Extended finite state machine. In: Specification of Software Systems. Springer, London (2011). https://doi.org/10.1007/978-0-85729-277-3_7
3. Aldrich, J., et al.: Permission-based programming languages (NIER Track). In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp. 828–831. ACM (2011). https://doi.org/10.1145/1985793.1985915
4. Arduino: Arduino libraries (2024). https://www.arduino.cc/reference/en/libraries/
5. Bacchiani, L., Bravetti, M., Giunti, M., Mota, J., Ravara, A.: A Java typestate checker supporting inheritance. Sci. Comput. Program. **221**, 102844 (2022). https://doi.org/10.1016/j.scico.2022.102844
6. Bachin, A.: Better enums (2015–2019). http://aantron.github.io/better-enums/
7. Balogh, R., Obdržálek, D.: Using finite state machines in introductory robotics. In: Lepuschitz, W., Merdan, M., Koppensteiner, G., Balogh, R., Obdržálek, D. (eds.) RiE 2018. AISC, pp. 85–91. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-97085-1_9
8. Barr, M.: Programming Embedded Systems in C and C++. O'Reilly (1999)
9. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API protocol checking with access permissions. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 195–219. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03013-0_10
10. Biffle, C.L.: The Typestate Pattern in Rust (2019). http://cliffle.com/blog/rust-typestate/
11. Bispo, J., Paulino, N., Sousa, L.M.: Challenges and opportunities in C/C++ source-to-source compilation. In: Bispo, J., Charles, H.P., Cherubin, S., Massari, G. (eds.) Proceedings of the 14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023). Open Access Series in Informatics (OASIcs), vol. 107, pp. 2:1–2:15. Schloss Dagstuhl — Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/OASIcs.PARMA-DITAM.2023.2
12. Bombarda, A., Bonfanti, S., Gargantini, A.: Developing medical devices from abstract state machines to embedded systems: a smart pill box case study. In: Mazzara, M., Bruel, J.-M., Meyer, B., Petrenko, A. (eds.) TOOLS 2019. LNCS, vol. 11771, pp. 89–103. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29852-4_7
13. Bonfanti, S., Gargantini, A., Mashkoor, A.: A systematic literature review of the use of formal methods in medical software systems. J. Softw. Evol. Process **30**(5), e1943 (2018). https://doi.org/10.1002/smr.1943
14. Boost Developers: Boost C++ libraries (Boost.Hana documentation). https://www.boost.org/doc/libs/1_84_0/libs/hana/doc/html/index.html
15. Boost Developers: Boost C++ libraries (2024). https://www.boost.org/
16. Caminiti, L.: Boost.Contract 1.0.0. https://www.boost.org/doc/libs/1_80_0/libs/contract/doc/html/index.html (2008–2019)
17. Clark, C., Kappert, E.: Openpyxl — a Python library to read/write Excel 2010 xlsx/xlsm files (2024). https://pypi.org/project/openpyxl/

18. Coblenz, M., et al.: Obsidian: typestate and assets for safer blockchain programming. ACM Trans. Program. Lang. Syst. **42**(3) (2020). https://doi.org/10.1145/3417516
19. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. SIGPLAN Not. **36**(5), 59–69 (2001). https://doi.org/10.1145/381694.378811
20. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24851-4_21
21. Driscoll, P.J., Parnell, G.S., Henderson, D.L.: Decision Making in Systems Engineering and Management. Wiley, Hoboken (2022)
22. Estivill-Castro, V., Hexel, R.: Run-time verification of regularly expressed behavioral properties in robotic systems with logic-labeled finite state machines. In: Proceedings of the IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), pp. 281–288. IEEE (2016), https://doi.org/10.1109/SIMPAR.2016.7862408
23. Formal Methods and SE Laboratory University of Milan, Formal Methods and Software Engineering Lab University of Bergamo: Asm2c++ (2006–2022). https://asmeta.github.io/download/asm2c++.html
24. Formal Methods and SE Laboratory University of Milan, Formal Methods and Software Engineering Lab University of Bergamo: Asmeta framework (2006–2022). https://asmeta.github.io/
25. Giampaolo, F.: psutil (2024). https://pypi.org/project/psutil/
26. Giftthaler, M., Neunert, M., Stäuble, M., Buchli, J.: The control toolbox — an open-source C++ library for robotics, optimal and model predictive control. In: Proceedings of the IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), pp. 123–129. IEEE (2018). https://doi.org/10.1109/SIMPAR.2018.8376281
27. Graphviz Team: Graphviz (2021). https://www.graphviz.org/
28. Gregor, D., Järvi, J.: Variadic templates for C++. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 1101–1108. ACM (2007). https://doi.org/10.1145/1244002.1244243
29. Hoang, T.S., Snook, C., Ladenberger, L., Butler, M.: Validating the requirements and design of a hemodialysis machine using iUML-B, BMotion studio, and cosimulation. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 360–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_31
30. Hu, Y., et al.: Decision-making system based on finite state machine for low-speed autonomous vehicles in the park. In: IEEE International Conference on Real-time Computing and Robotics (RCAR), pp. 721–726 (2022). https://doi.org/10.1109/RCAR54675.2022.9872208
31. Hwang, S., Lee, K., Jeon, H., Kum, D.: Autonomous vehicle cut-in algorithm for lane-merging scenarios via policy-based reinforcement learning nested within finite-state machine. IEEE Trans. Intell. Transp. Syst. **23**(10), 17594–17606 (2022). https://doi.org/10.1109/TITS.2022.3153848
32. Jakobsen, M., Ravier, A., Dardha, O.: Papaya: global typestate analysis of aliased objects. In: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP). ACM (2021). https://doi.org/10.1145/3479394.3479414
33. Kitware Inc.: Cmake (2024). https://cmake.org/

34. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 146–159. ACM (2016). https://doi.org/10.1145/2967973.2968595
35. Kumar, B.: A survey of key factors affecting software maintainability. In: Proceedings of the International Conference on Computing Sciences, pp. 261–266 (2012). https://doi.org/10.1109/ICCS.2012.5
36. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383
37. LLVM Developer Group: libclang-17-dev: Development package for Clang (2024). https://packages.debian.org/search?keywords=libclang-17-dev
38. Mayr, M., Salt-Ducaju, J.M.: A C++ implementation of a cartesian impedance controller for robotic manipulators (2022)
39. McKinney, W., et al.: pandas: powerful Python data analysis toolkit (2024). https://pandas.pydata.org/
40. Militão, F., Aldrich, J., Caires, L.: Substructural typestates. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV), pp. 15–26. ACM (2014). https://doi.org/10.1145/2541568.2541574
41. Mota, J., Giunti, M., Ravara, A.: Java typestate checker. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 121–133. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_8
42. Oliveira, M.F., Redin, R.M., Carro, L., Lamb, L.D.C., Wagner, F.R.: Software quality metrics and their impact on embedded software. In: Proceedings of the 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), pp. 68–77. IEEE (2008). https://doi.org/10.1109/MOMPES.2008.11
43. Porkoláb, Z., Mihalicza, J., Sipos, A.: Debugging C++ template metaprograms. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE), pp. 255–264. ACM (2006). https://doi.org/10.1145/1173706.1173746
44. Python Software Foundation: time — time access and conversions (2024). https://docs.python.org/3/library/time.html
45. Redin, R.M., et al.: On the use of software quality metrics to improve physical properties of embedded systems. In: Kleinjohann, B., Wolf, W., Kleinjohann, L. (eds.) DIPES 2008. ITIFIP, vol. 271, pp. 101–110. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-09661-2_10
46. Reis, G.D., J. D. Garcia, J. Lakos, A.M., N. Myers, B.S.: Support for contract based programming in C++ (2018). https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0542r5.html
47. Rust Language: The Embedded Rust Book (2018). https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html#typestate-programming
48. Sherman, T.: Quality attributes for embedded systems. In: Sobh, T. (ed.) Advances in Computer and Information Sciences and Engineering, pp. 536–539. Springer, Dordrecht (2008). https://doi.org/10.1007/978-1-4020-8741-7_95
49. Šimoňák, S.: Verification of communication protocols based on formal methods integration. Acta Polytechnica Hungarica **9**(4), 117–128 (2012). http://acta.uni-obuda.hu/Simonak_36.pdf
50. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **SE-12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929

51. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, E.: First-class state change in plaid. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 713–732. ACM (2011). https://doi.org/10.1145/2048066.2048122
52. TIOBE: TIOBE Index (2000–2023). https://www.tiobe.com/tiobe-index//
53. Tolmer, V.: Protenc library (2019). https://github.com/nitnelave/ProtEnc
54. Wang, J., Tepfenhart, W.: Formal Methods in Computer Science. Chapman and Hall/CRC (2019). https://doi.org/10.1201/9780429184185
55. Xiao, X., Balakrishnan, G., Ivančić, F., Maeda, N., Gupta, A., Chhetri, D.: Arc++: effective typestate and lifetime dependency analysis. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 116–126. ACM (2014). https://doi.org/10.1145/2610384.2610395
56. Yin, T.: Lizard: an extensible cyclomatic complexity analyzer (2024). https://pypi.org/project/lizard/1.8.7/