UPPSALA
UNIVERSITET

UPTEC F 23044
Examensarbete 30 hp
Juni 2023

State Machine Model-To-Code Transformation In C

Jonathan Carlgren
Per William Oskarsson

Civilingenjörsprogrammet i teknisk fysik

UPPSALA
UNIVERSITET

State Machine Model-To-Code Transformation In C

Jonathan Carlgren
Per William Oskarsson

Abstract

A state machine model can turn a complex behavioural system into a more accessible graphical model, and can improve the way people work with system design by making it easier to communicate and understand the system. The clear structure of a state machine model enables automatic generation of well structured, and consequently readable, and maintainable code. There are many known implementations and even plenty of commercial software available for working with state machines, and the goal of this project is to compare and discuss some of these implementations in the context of Ericsson's demands for run time, memory usage, scalability, readability, and maintainability.

More specifically, the project focuses on the state machine models specified in the UML (the unified modeling language [15]), utilizing UML's associated markup language, XMI, to go from graphical model to generated C code. The resulting C code is primarily a code skeleton which only provides the basic behaviour of transitioning between states given a specific event, it is expected that the developers manually implement additional features themselves. The examined implementations are: Nested Switch, Array of Structs, Function Pointers, Basic State Pattern, State-Table Pattern, and Hierarchical State Pattern. Additionally, the project investigates how multiple state machines can communicate and work together as interacting state machines. And finally, to showcase how a state machine implementation can be maintainable, we develop an iterative code editor that can edit already operating and manually modified state machine implementations.

The implementations are tested on a case study example provided by Ericsson, aimed to represent a sort of typical state machine design when it comes to number of states and events. The implementations are further tested with randomly generated state machines, to examine their scalability properties.

In our opinion the results favour the Array of Structs and Basic State Pattern implementations and the choice depends on the optimisation used and the priority between run time and memory.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Lisa Granholm Ämnesgranskare: Mikael Sternad

Examinator: Tomas Nyberg

Populärvetenskaplig sammanfattning

För systemutveckling inom inbyggda system används tillståndsmaskiner vanligen som verktyg för att modellera komplexa beteenden. Det är ett enkelt sätt att få en överblick av systemet som helhet samt lägga till funktionalitet. Dessa system kan ta lång tid för utvecklare att från en specifikation modellera dessa som en tillståndsmaskin till att sedan implementera den i kod. Forskning inom modellbaserad utveckling belyser olika vägar på hur man kan översätta dessa modellerade tillståndsmaskiner direkt till kod, för att automatisera detta arbete så mycket som möjligt. I detta examensarbete föreslås ett sätt att arbeta med tillståndsmaskiner, från UML modell till C kod genom att utveckla kodgeneratorer. Kodgeneratorerna skapar kodskelett för tillståndsmaskiner enligt sex olika designsätt från vetenskapliga artiklar och implementationer online. De genererade kodskeletten utvärderas utifrån körtid, minnesanvändning, skalbarhet, läsbarhet samt underhållsmässighet. Efteråt utökas arbetet genom att undersöka olika sätt att modellera kommunikation mellan två tillståndsmaskiner som körs sekventiellt. En av dessa kodgeneratorer utökas även med funktionalitet för att kunna uppdatera en tillståndsmaskin i redan genererad kod som ett "proof of concept".

Implementationerna testas både på en given tillståndsmaskin och spontant genererade tillståndsmaskiner, där resultaten för implementationerna använda föreslår att "Array of Structs" och "Basic State Pattern" presterar bäst. För kommunikation mellan två genererade tillståndsmaskiner så är ett resultat svårare att urskilja. Den extra implementerade funktionaliteten för att iterativt uppdatera en genererad kod för en tillståndsmaskin fungerar väl och är ett sätt att göra mer underhållsmässig kod.

Acknowledgements

We would like to forward special thanks to our subject reviewer Mikael Sternad for helping us out with the thesis. We would also like to thank our supervisors Mattias Hellsing and Lisa Granholm and the rest of the Arkopool team at Ericsson for giving us the opportunity to work with this thesis.

Contents

1 Introduction

1.1 Background

Embedded systems exist everywhere where electronics are present, in all type of appliances whether it be your refrigerator or mobile phone. These devices can be complex systems loaded with software. One challenge in software development is how to best represent these complicated systems in code. A widely used practice is to model the systems as state machines that consists of different states and transitions. In software development, a lot of time can be spent translating system specifications from simple text or a model language, into source code. Therefore, a great amount of research has gone into automatically generating state machines from modeling language into code.
The concept of automatically generating source code from a state machine model is largely a solved equation, with many modeling tools offering code generation in their applications. However, in many cases the code generated by these tools may not be efficient for the specific system at hand. For systems with more advanced features such as hierarchy or history, the generated code might not even support the functionality correctly.

1.2 Ericsson

Ericsson is a Swedish multinational company with around 100 000 employees worldwide and has been a major contributor in the networks and telecommunications industries during the 147 years of its existence. Embedded systems are a fundamental part of these industries, and a good understanding of state machines can be a great tool to streamline the software development process for companies such as Ericsson.

1.3 Problem statement

As it stands currently, the division at Ericsson has no centralised way of working with state machines. It usually boils down to getting a specification explained in simple text and then implementing the functionality manually. Thus, they would like a showcase of how the general workflow could look like when working with state machines, if possible as automated as can be. There are a wide variety of tools that can model state machines and also generate code from the model, however these would have to be put through security checks and they more or less follow a similar set of generalized coding. A prospect of interest was therefore to study some of the existing research of state machine code generation and compare the performance of some different implementations. The thesis will mainly aim to cover the following topics:

- Suggest a general way to work with state machines, from model to C code.
- Investigate current design practices for state machines and compare how they perform against each other.
- Implement support for iteratively changing already generated state machine code.
- Investigate design practices for interacting state machines.

The thesis work begins with a literature study to better understand the current research and how state machines are used in practice. From the literature study, six different state machine implementations are selected to be implemented from model to code. The implementations will be tested and evaluated with both randomly generated state machines, and a case study provided by Ericsson. Finally, the work is extended to include interacting state machines and an editor that can iteratively update already generated state machine code.

1.4 Division of Labour

Both of us have conducted our own literature studies to gather information, and from there on we chose three state machine implementations each. Per is responsible for the three state pattern approaches, while Jonathan is responsible for the other three. For the additional extensions, Per looks at the state machine interaction sections, and Jonathan the iterative code editor parts.

2 Theory

This section introduces the concepts necessary to understand the complete process of going from a state machine model to functioning state machine code.

2.1 State Machine

A state machine is a mathematical modeling concept, where the behaviour of a system is described by states, and transitions between states. It is commonly used to express the intended behaviour of electrical engineering and software systems, to help reduce complexity and provide a graphical overview of the systems. The concept is suitable for systems involving user interfaces (such as Ericsson's) that should trigger transitions in response to external events [3]. Many examples can be taken from electrical engineering systems where for example a power switch can be described as a state machine including two states "on" & "off" with a transition between the states triggered by an external event, "press power button".

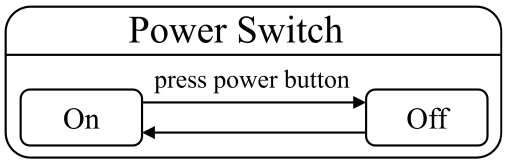


Figure 1: Power switch state machine example.

We will call this type of state machine a "flat state machine", meaning a state machine where the transition only can depend on one external event and the current state. For some more complex software systems, additional features are required for the state machine in order to fully describe the behaviour of the systems. Some of these features were introduced by David Harels "Statecharts", during his work with the Israel Aircraft Industries that started in 1983 [4]. Statecharts extends state machines with features such as "nested states", "parallel states", and "history states", which are essential to describe the behaviour of certain systems [19].

2.2 UML

Working with model driven development has become an essential tool for many large scale software development projects. To ensure scalability, security and robustness, it is important to have a clear architecture for the modeling language. Rational Software began developing The Unified Modeling Language (UML) in 1994 to standardise the way system architects, software engineers, and software developers work with modeling to ensure a clear architecture [24, 17, 9]. UML includes many different types of models and state machine models are one of the behavioral models included. In 1997, UML was adopted by the Object Management Group (OMG), a non-profit software consortium and standards development organisation backed by prominent technology organisations, making UML an industry standard for system modeling and design [15, 16].

2.3 UML State Machine

At the time of this report the latest UML version is UML 2.5.1, and its specifications can be found at UML.org [17]. We mostly focus on flat state machines in this report (since it is the type of the case study provided by Ericsson), but we also compare the performance of a more advanced implementation. The additional state machine features included in this implementation are introduced below.

2.3.1 Hierarchy and composite state

One significant feature is hierarchy, it allows a substate to inherit characteristics from a parent state, and enables the existence of composite states. Composite states contains one or more regions within itself, and allows for orthogonal independent behaviour within each region. The regions may include further hierarchy with additional states, composite states or even state machines within them. In Figure

2, the composite state "Computer Science I" contains two orthogonal regions with additional states and transitions within them.

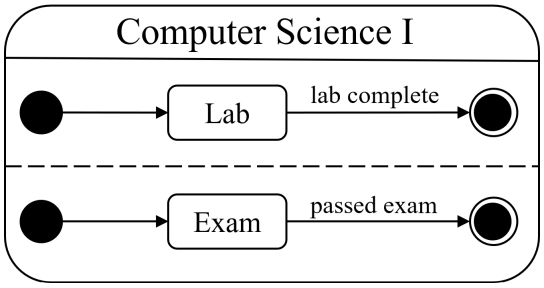


Figure 2: Composite state "Computer Science I" with two orthogonal regions containing the substates "Lab" and "Exam". The dashed line indicates orthogonality between the regions, meaning that progress within the two regions are independent of each other.

2.3.2 History state

There is support for two types of history states in UML, shallow and deep history. Shallow history remembers the topmost substate configuration, while deep history remembers the full state configuration. When exiting a region within the state machine, the history states can remember the state configuration within that region allowing the state machine to return to the configuration later on.

(a) Shallow history

(b) Deep history

Figure 3: History pseudostates.

2.4 XML and XMI

The Extensible Markup Language (XML) is one among file formats like JSON, YAML and HTML that can be used to store data. There are different benefits of using the different types to store a representation of a state machine, but what makes XML desirable is the focus on structure and readability. It is a format widely used and as such, there are many available tools that handles XML files.
For UML diagrams in particular the Object Management Group (OMG) have developed an application of the XML format tuned for UML specifications, namely the XML Metadata Interchange specification [14]. The specification uses convenient conventions for naming attributes of the different state machine elements such as "uml:State", "uml:Transition" and has attributes giving these elements a unique id, "xmi:id". This makes the XMI format easy to parse.

2.5 Modeling Software

There are many different tools that support modeling with UML and XMI, both open source and commercial. Some examples include Visual Paradigm, Eclipse Papyrus, and Microsoft Visio [7, 18, 1].

2.6 Model-To-Code Generation

Model transformations is an important aspect in Model-Driven Engineering (MDE) [11, 21]. It is a framework that sets the rules for how to automatically transform a higher-level model, into another model (Model to Model transformation) or source code (Model-to-Code transformation) for example.
To achieve Model-To-Code transformation there are many approaches, domain-specific languages (DSL) such as WebDSL or machine learning based algorithms to name a few.
A simple and common proposal is to generate the code using a one to one mapping of state machine

components to appropriate data structures in the code language of choice. For advanced systems this might not be enough, but for the analysis conducted here, it will suffice.

2.7 Code Generator Design

Designing the code generator will be broken down into two parts - creating a parser and generating the code itself. A parser reads through data and breaks it down into recognized characters. For this context, the parser should read the modeled state machine specification and save the contents of the state machine like states, transitions and so on. The code generation step uses the parsed data and writes it into a .c code file and if desired, a .h header file to create working C code for a state machine following the model specification.

2.8 State Machine Implementations

This section showcases the different ways of implementing a state machine that will be compared. The code generated from UML models will follow these designs, which can be broken down into pattern-based designs and non pattern-based design [6, 10]. A design pattern in software can be explained as a description of how to implement a system or application following an established software design pattern first introduced by the Gang of Four consisting of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [10]. They provide a general design solution, but do not specify details like how to handle transitions and model semantics. That is up to the developer. Non-pattern based designs do not follow any established pattern.

2.8.1 Nested Switch/If Statements

One of the most common and straight forward approaches is using nested switch/if statements. It provides an effective solution for simple flat state machines, but lack support for more advanced state machine features [22, 6]. The outer switch/if statement keeps track of the current state as a scalar variable, while the inner switch/if statement keeps track of the current event as a scalar variable. The current event is provided to the system by some external event.

2.8.2 Array of Structs

Another way of implementing a flat state machine is using an array of structs. Since it is a straightforward approach the exact origin of this approach is unclear, but our implementation is based on an article by Amlendra Kumar [2]. The structs contains possible combinations of states and events, and the indexing of the structs is based on enumerates of possible states or/and events. There are different ways of ordering the array depending on how the user intends to work with the state machine model.

2.8.3 Function Pointers

The Function Pointers implementation stores pointers to transition functions in a 2D array with the number of states and events making up the dimensions of the array. This is one of many different implementations that utilises a state-event-table approach [22]. Similar to the Array of Structs approach, we enumerate all possible states and events and the current state and event corresponds to the index of the function pointer to the correct transition function. The array size grows quickly with number of states and events, however the array will be sparse and the look up time is independent of array size.

2.8.4 State Pattern

One of the behavioral patterns in software design is the state pattern. It was originally introduced in 1994 by the Gang of Four ([10]) and has since then become a mainstay in state machine design, being the base for several pattern-based extensions of state machine representation in code [13, 20, 22, 23].

The main idea behind the state pattern is to allow an object to change its behavior when it changes internal states. In this way, the behavior for a state machine can change depending on the current state. States should be independent of each other so that implementation of new states does not affect the

previously defined ones, which makes the code easier to maintain.

A state machine implemented with the state pattern design works as a context object that contains a reference to the active state object. The context communicates and delegates work to the active state through an interface. This interface defines state-wide methods like entry and exit actions and setting the state. Individual states are used for the state-specific behavior. What the state pattern does not tell one however, is how transitions are handled and is up to the specific implementation to decide.

2.8.5 State-Table Pattern

The State-Table pattern is a variant of the State Pattern that handles transitions using a table, such as a matrix or hash table for instance. This is as opposed to most traditional State Pattern implementations, that handles transition through class methods. A table-based approach could be desirable for systems with many states and transitions, as it brings more structure where code management using other approaches may become difficult.

2.8.6 Hierarchical State Pattern

The traditional approaches to implement state machines through pattern and non-pattern based design have their benefits, but are limited in supporting the full functionality that UML state machines can provide. One of such functionalities are hierarchical states. While generating code for hierarchical state machines in particular is not the main interest of the project, it can be good to see how well an implementation with more features performs compared to the more simpler approaches. There have been several proposals for implementations to support different or all parts of UML models.

In the proposal by Sunitha and Samuel ([23]), code skeletons for a state machine are generated in Java from UML State Charts with support for hierarchy, concurrency along with history. They provide a template for their pattern in Figure 4. As with the regular State Pattern, Samuel and Sunitha propose each state and event as distinct classes, along with a context class keeping track of the active state and delegating the events to the states. State hierarchy is supported by using inheritance in a composite state class that derives the substates. To represent concurrency, one can have parallel regions in this composite state by the use of the `OrthogonalProperty` class that is equipped with methods to set regions and handle subtransitions. While shallow history is stored in the context class, deep history is its own class and is managed by the composite state and `OrthogonalProperty` class as well. When comparing implementations, this implementation will be named "Hierarchical State Pattern" so that it can be referred to easier and not be confused with the other state pattern-based approaches.

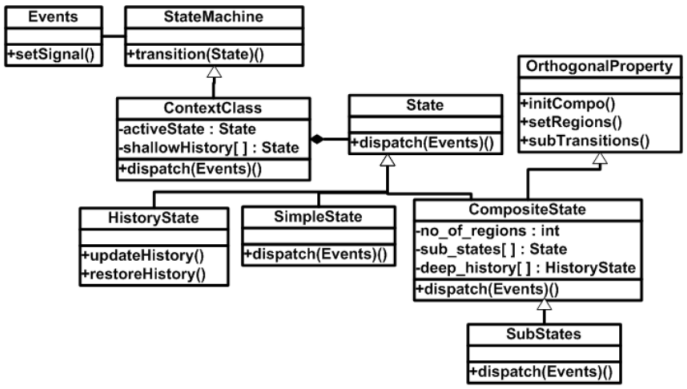


Figure 4: Proposed pattern template by Sunitha and Samuel [23]

.

2.9 Interacting State Machines

Another area of state machine design that was of interest was how to implement state machine interaction. There can be different desired goals of communication to compare, and in this thesis we limit the scope to the transition logic of the state machines. Namely, when the transition of a state depends on the state of another state machine. Large systems consists of several state machines with their distinct functions that interact with one and another and exchange data, which makes it important that the interaction between them is handled efficiently. The systems used by the division at Ericsson have the individual interacting state machines running sequentially. As such, the implementations analyzed here will not be using any parallelization methods even though they could very well be useful in other settings.

2.9.1 Switch statement with global variables

A simple idea is to only add conditional statements into the event handlers of the state machines that checks the active state of the other state machine. This requires the context/state machine object of the other state machine to either be global or sent in as additional input which can be less desirable. As a result, while having low communication overhead compared to other approaches, the code might become harder to read and maintain for larger state machines. To reference this method easier, we call it the "Global Switch" approach.

2.9.2 Function Pointers

Similar to implementing regular state machines, a lookup table with function pointers to handle the transition logic could be expanded to cover different cases when the state machine needs to take another action depending on another state machine. A suitable approach is to use a 3D-array. Each event consists of a cell in the 3D-array, which points to a matrix, where each matrix element is a combination of the states of the two state machines for that event. If there is a different action to be taken for a select set of states, that element will point to the other state to transition to. Using a function pointer array can be more efficient, however for large applications these arrays cost increasing amounts of memory. A 3D-array uses even more memory than the regular state-event lookup table.

2.9.3 Message Passing

Message passing can be used for most systems using state machines. It works well in parallel state machines, or in distributed and event-driven systems, for instance. A system using message passing creates a message with information and sends it to a process. That process can then use that information to take the relevant action. An example is when a state machine performs a transition to a new state. The new active state is sent in a message to the other state machine that is handling an event and will transition differently depending on the active state read in the message.

2.9.4 Mediator Pattern

The mediator pattern is a behavioral design pattern that is used to reduce communication complexity between objects. Instead of objects being dependent on multiple different classes, all communication is handled through a mediator class by having the communicator contain references to the state machines. This decouples components which can make code more maintainable, since communication is centralized [12].

2.10 Iterative Code Editor

An additional feature requested by Ericsson is to make the generated code updatable. Since the initial code generation only results in a code skeleton and the user usually needs to add specific behaviour manually, it is important to be able to keep the manually written code when updating the state machine model. This can be done using Python string manipulation where a python script can read, navigate, edit, and write to a previously generated state machine implementation file. The distinct structures of some of the implementations makes it easy to navigate the file and find the specific code elements that

should be edited without touching the manually added code. The ambition is to enable insertion and extraction of states, events, and attributes related to the two.

2.11 Test Case

The state machine implementations are first tested on a case study of a state machine provided (and anonymized) by the division at Ericsson, which can be seen in Figure 5. It is a state machine with many transitions compared to states. This case study is modeled using a modeling language (in our case Papyrus and Visual Paradigm Community) and stored as an XML/XMI file. The file is then run through the parser, and finally the code skeletons are generated according to the different designs.

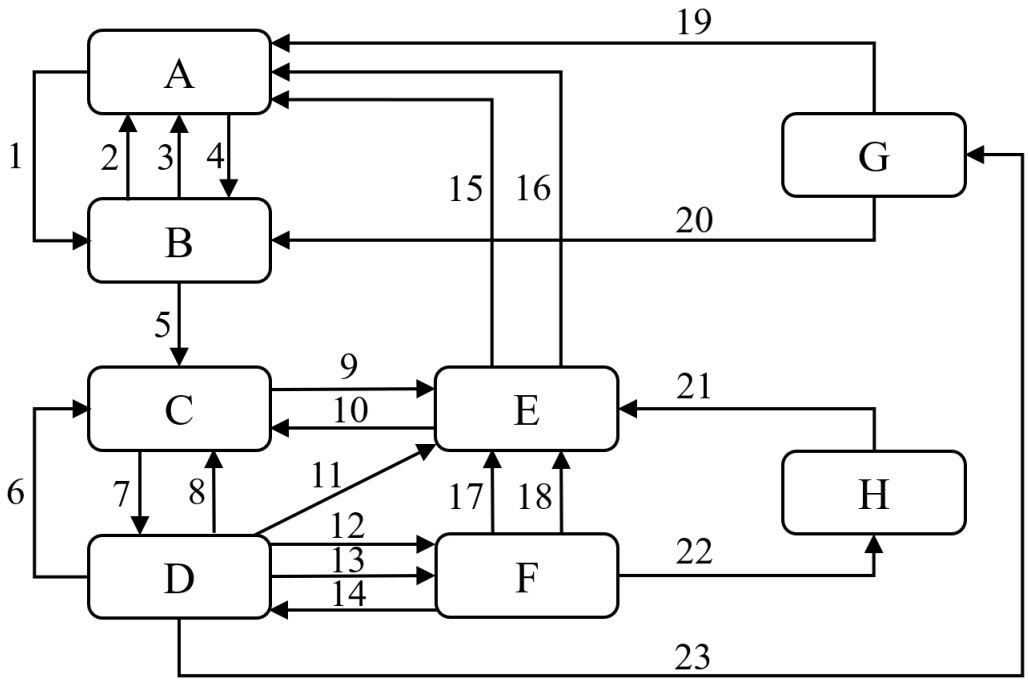


Figure 5: State machine model of the case study.

2.12 Performance Parameters

The following section brings up the parameters used to estimate the performance of the implementations. These parameters consider how efficient the system runs overall and also looks at some basic software design principles.

2.12.1 Run Time

Run time is a common metric of efficiency. It is important that the state machine has good throughput as transitions in modern systems only takes nanoseconds.

2.12.2 Memory usage

Depending on where the software is mounted, having low memory usage can be a must. For small integrated circuits in embedded systems for example, the available memory is often quite low, which makes it imperative for the software to keep the memory consumption at a minimum.

2.12.3 Scalability

Scalability of an implementation is essential to maintain optimal performance as the state machine grows. It measures how well performance scales with increasing memory demand. For large systems in the industry, you want to add states and transitions to a state machine without a great decrease in throughput.

2.13 Code metrics

The following metrics are ones that are either impossible to measure or have no clear way to be measured. They consist of important qualities in code design that is relevant in production. As such, they will not be measured, but commented on.

2.13.1 Maintainability

Maintainability is an important metric in software design. The generated code should be easy to understand, change and test. Code that is too complex or require large amounts of micro modifications in different places is expensive, and less likely to be used as of it in production.

2.13.2 Modularity

Modular code has its functionality split into independent parts. This makes the code more organized, and easier to maintain and test.

3 Implementation

This section aims to give an overview of how one can work with state machines, from being given a specification, to modeling the specification and finally arrive at a desired code skeleton. Additionally, different types of state machine implementations that can be used as framework for code generation is highlighted. These are implementations found in articles of related work, or generally established ideas that were discussed in the previous section.

3.1 Modeling

To model the UML state machines in this thesis, Papyrus and Visual Paradigm Community Edition was used. Papyrus saves UML models as .uml files, and Visual Paradigm is able to convert the model into XML and XMI, which makes them suitable for this application.

3.2 Parsing

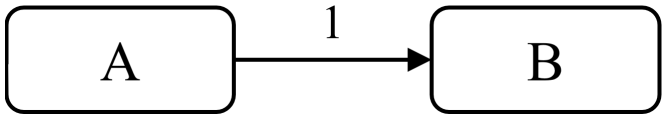
Parsing of the UML files is done in a Python script using the built-in ElementTree XML API [8]. It parses a given XML file and represents it as a tree structure, exploiting the hierarchical structure of XML data. Then, given a state machine diagram saved as a .uml file, the parser iterates through the elements of the tree until it finds nodes with attributes "uml:State" for states, "uml:Transition" for transitions, and more. The relevant data of these nodes are stored in separate dictionaries.

3.3 Code Generation

The code generator takes the parsed data and uses it to write the header and code skeleton for the state machine, following one of said implementations. It is accomplished in the script by iterating through the data in the dictionaries, and writing the code automatically in a string literal which is then written into the target source code .c file and header .h file.

3.4 `eventHandler` Functions

In the Nested Switch, Function Pointers, and Array of Structs implementations, the transition logic is implemented using `eventHandler` functions. Initially, the `eventHandler` functions only contains the basic transition logic of returning the next state, see Figure 6. The purpose of these functions is to facilitate manually adding new features for specific transitions in a structured way.



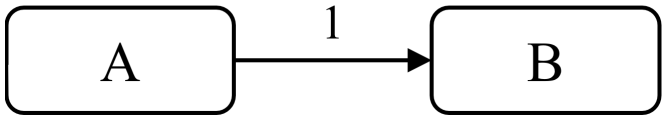

```
1 enumStates Event1Handler(void)
2 {
3         return StateB;
4 }
```

Figure 6: A simple state machine model consisting of two states "A" & "B", and one event "1", and the generated `eventHandler` function related to the transition.

3.5 Nested Switch

The Nested Switch approach creates enumerated types for all possible states and events, it then utilizes nested switch statements to find specific `eventHandler` functions and consequently trigger transitions between states. Whenever a new event occurs the state machine finds the combination of the current

state in an outer switch statement and the new event in an inner switch statement. It proceeds to call an `eventHandler` function that handles the transition and returns the next state. In the basic code skeleton the `eventHandler` functions exclusively returns the next state, with further functionality left for the user to implement manually. Figure 7 illustrates a simple state machine model and a fragment of the resulting generated code with the Nested Switch logic.

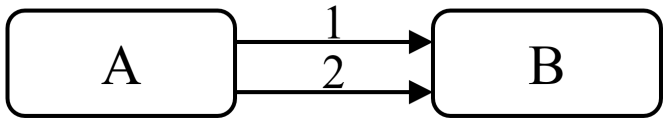



```
1  ...
2  switch(state)
3  {
4  case StateA:
5  {
6      switch(event)
7      {
8      case Event1:
9      {
10         nextState = Event1Handler();
11         break;
12     }
13     }
14     break;
15 }
16 case StateB:
17 ...
```

Figure 7: A simple state machine model consisting of two states "A" & "B", and one event "1", and a fragment of the resulting generated code with the Nested Switch logic.

3.6 Function Pointers

The Function Pointers approach creates enumerated types for all possible states and events, and a 2D array of function pointers that point to `eventHandler` functions. The size of the enumerate types determine the dimensions of the array, and when the state machine is running, the index of the current state and the new occurring event corresponds to a specific function pointer in the array. The resulting `eventHandler` function returns the next state. Figure 8 illustrates a simple state machine model and a snippet of its generated code skeleton using the Function Pointers approach.

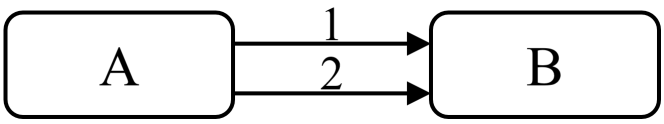


```
1 ...
2 static eventHandler StateMachine =
3 {
4     [StateA] = {[Event1] = Event1Handler, [Event2] = Event2Handler},
5     [StateB] = {}
6 };
7 ...
8 nextState = (*StateMachine[currentState][newEvent])();
9 ...
```

Figure 8: A simple state machine consisting of two states "A" & "B", and two events "1" & "2". The code snippet contains the 2D array "StateMachine" starting at line 2, with the number of states corresponding to the first dimension and the number of events corresponding to the second dimension of the array. Line 8 shows how the call to the `eventHandler` function is made from the 2D array.

3.7 Array of Structs

The Array of Structs approach creates enumerated types for all possible states and events, and a struct type that contains a state, an event, and a pointer to an `eventHandler` function. It generates an array of these structs with all the valid combinations of states and events given in the state machine, with the enumerated events corresponding to arrays indices. When the state machine is running the enumerate value of a new occurring event corresponds to the index of the corresponding `eventHandler` function, which in turn, returns the next state. Figure 9 illustrates a simple state machine model and a snippet of its generated code skeleton using the Array of Structs approach.




```
1  ...
2  typedef struct
3  {
4          enumStates stateMachineState;
5          enumEvents stateMachineEvent;
6          eventHandler stateMachineEventHandler;
7  } stateMachineStruct;
8  ...
9  stateMachineStruct stateMachine[] =
10 {
11         {StateA, Event1, Event1Handler},
12         {StateA, Event2, Event2Handler}
13 };
14 ...
15 nextState = (*stateMachine[newEvent].stateMachineEventHandler)();
16 ...
```

Figure 9: A simple state machine consisting of two states "A" & "B", and two events "1" & "2". The code snippet contains the struct declaration starting at line 2, the struct contains a state, an event and an `eventHandler` function pointer. The structs are kept in the array starting from line 9 and each event corresponds to one entry in the array. Line 15 shows how the call to the `eventHandler` function is made from the array.

3.8 OOP in C

Object-Oriented Programming (OOP) is a programming paradigm that focuses software design around objects and classes, as opposed to for example functional or imperative programming. Implementing state machines in code using OOP concepts can prove to be beneficial for pattern-based designs focusing on object encapsulation, such as the State Pattern, which is why a good amount of related work in the field writes code in popular OOP languages such as C++ or Java. The functionality that OOP brings is not natural to C, which means we have to emulate it if we want to use a similar design pattern. There are 4 important concepts that OOP consists of - polymorphism, inheritance, encapsulation and abstraction. The concepts that will mostly be of use for the State Pattern implementations are polymorphism and inheritance [5].

In C there is no such thing as a class, instead structures are used. They can hold data similar to class variables, and function pointers to emulate class methods. With class inheritance, a class inherits the features of another parent class. In C, a child can have an instance of its parent struct as the first member. The child struct is then able to access variables of the parent struct through pointer casting.
Polymorphism is the idea of multiple different objects having the same interface and use the same method calls, but implement them differently. In C, one could accomplish this using function pointers as class methods, and then pointing to different functions depending on the object. Each object then has its own version of the same method, pointing to different functions. It makes code more reusable and flexible.

3.9 Basic State Pattern

In the skeleton for the state pattern, each state is implemented as individual structs, which contains function pointers to event functions, emulating class methods. Additional distinct behavior of the states can be added into said structs. Every state has an interface struct inherited that has pointers to more generic statewide functions such as `setState()`. The overarching state machine object is defined as a context struct, which stores the current active state and inherits all the other distinct state structs. The generated code skeleton provides function bodies for the class methods of specific events, but the contents are left to be implemented by the user for their own specific state machine. One instance of this design is given in Figure 10. We will call this the "Basic State Pattern", in which transitions are called through the class methods of the states.

```
1  ...
2  struct State {
3          void (*setState)(State **source, State *target);
4          void (*entry)();
5          void (*exit)();
6  };
7
8  typedef struct {
9          State interface;
10         void (* T0_event)(Context *ct);
11          //Additional state behavior...
12 }S0State;
13
14 typedef struct {
15         State interface;
16         void (* T1_event)(Context *ct);
17          //Additional state behavior...
18 }S1State;
19
20 struct Context {
21         State *active;
22         S0State S0;
23         S1State S1;
24 };
25 ...
26 void S0_T0_event(Context *ct){
27         setState(&(ct->active), (State *)&S1);
28 };
29
30 void setState(State **source, State *target){
31         *source = target;
32 };
33 ...
```

Figure 10: Example of how a state machine can be designed using the State Pattern design. A state machine with states "S0" and "S1" are defined with their individual behavior along with the inherited statewide class methods in the "State" struct. The context object stores references to all states and the active state. An example of the transition logic is shown in the `S0_T0_event` function along with the interface `setState` function.

3.10 State-Table Pattern

This version of the state pattern is more akin to the Array of Structs idea. It stores states and events as enum variables, but still has the context struct to store the active state. The other struct is for the state-event table, storing an event and a pointer to a state handler. The generated code skeleton creates the state-event tables where the rows are ordered by states and columns by events. An entry in the table that corresponds to an event happening while the state machine is in the current state will return a struct, where the state handler field is a function pointer to a state handler function. In the state handler functions the user can implement the distinct behavior of the state, following the state design pattern. An example is given in Figure 11.

```
1  ...
2  void S0handler(Context *ct, Event e){
3          //Define state behavior...
4          ct->active=S0;
5  };
6
7  void S1handler(Context *ct, Event e){
8          //Define state behavior...
9          ct->active=S1;
10 };
11
12 static EventTable transitiontable[nostates][noevents] = {
13         //S0
14 {
15         {T0,S1handler},
16 },
17         //S1
18 {
19         {T1,S0handler},
20 },};
21
22 void eventfunc(Context *ct, Event e){
23         EventTable transition = transitiontable[ct->active][e];
24         statehandler sh = transition.sh;
25         sh(ct, e);}
26 ...
```

Figure 11: Code example of using the State-Table Pattern implementation. Individual state behavior of states "S0" and "S1" is encapsulated in their respective handler functions in lines 2 and 7. The transition table is defined in line 12, with an event along with the active state being one entry in the table. The state handler function is accessed from the struct in "eventfunc" in line 22.

3.11 Hierarchical State Pattern

Following the pattern proposed by Sunitha and Samuel ([23]) structs are defined as in Figure 4. As with the regular state pattern, the states are expressed as distinct structs. There is a context struct that stores the active state, shallow history state and inherits a state machine struct which models the state machine itself. It contains a pointer to a dispatch function which delegates events to the states. To handle deep history, the `HistoryState` struct is used, which inherits the state interface and stores pointers to the `updatehist()` and `restorehist()` functions. A composite state takes the form of the `CompositeState` struct keeping track of the number of regions of said state and arrays of the substates and deep history states nested in it. For concurrency, they suggest the `OrthogonalProperty` interface that employs methods to set the number of regions and handle transitions between substates within regions. The composite state inherits this class. For a flat state machine without the need of the composite state functions of the implementation, an example is given in Figure 12.

```
1  ...
2  void Context_dispatch(Context *ct, Event *e){
3          dispatch[ct->active->stateid](ct, e);
4  };
5
6  void S0_dispatch(Context *ct, Event *e){
7          switch(e->sig){
8          case T0_signal:
9  {
10         ct->sm.transition = T0_transition;
11         ct->sm.transition(ct, (State *)&S1);
12         break;
13 }
14         case T1_signal:
15 {
16         ct->sm.transition = T1_transition;
17         ct->sm.transition(ct, (State *)&S2);
18         break;
19 }
20         default:
21 {
22         break;
23 }
24 } };
25
26 void T0_transition(Context *ct, State *target){
27         //Do something
28         ct->active = target;
29 };
30 ...
```

Figure 12: Code fragment of a flat state machine implemented using the Hierarchical State Pattern implementation. The context object "ct" of the state machine object "sm" delegates an incoming event through the active state `Context_dispatch` function in line 2 by calling the appropriate dispatch function. If `S0_dispatch` is called it checks the event signal and makes the parent state machine object transition to the next state.

3.12 State Machine Interaction

The different proposals for state machine interaction support (except for the 3D array approach which is more of its own implementation) is added into the Basic State Pattern approach described in the State Pattern section. This is because we are mainly interested in comparing different state machine interaction methods in this section and not the state machine implementation in itself. The State Pattern implementation was used as the base state machine implementation for this mainly due to it showing good results from the state machine implementation benchmark and being well structured.

3.12.1 Global Switch

There are two global context objects modeling the state machines, storing the active state and keeping references of all states in their state machine. In the event handler functions e.g. `S0_T0_event()` from Figure 10, a switch statement checks the active state id of the other state machine and then performs the appropriate transition depending on the state using the `setState()` method in the state class.

3.12.2 Mediator Pattern

A `Mediator` struct stores the state machine context objects for each state machine, and all functions, such as the event handlers and `setState()` method is changed to only take the mediator as input. The event handlers then reads the active state id of the other context from the mediator through a switch statement and executes the transition needed.

3.12.3 Message Passing

Using message passing, the code makes use of `Message` structs which contains a pointer to the active state of the state machine sending it. Additionally, the context structs hold message queues along with the current queue size. In the event handlers, the first message in the queue is used to determine the appropriate transition together with the current state. Then, when the correct transition is made, a message with the new current state is sent to the other state machine using the extra function `msg_queue()`. Another function, `msg_dequeue()` removes the read message from the queue.

3.12.4 3D array

The code uses four different enum variables for two state machines covering the events and states for each: `M0States`, `M1States`, `M0Events` and `M1Events` (M0 and M1 is used as names for the state machines as an example). A 3D array is then defined for each of the state machines with the dimensions M0StatesxM1StatesxM0Events for M0 and M1StatesxM0StatesxM1Events for M1. If there is a valid transition to make for any combination, the array contains a function pointer to a state handler function taking care of the transition and other behaviour one would want to implement. Non-valid combinations are initialized as NULL.

3.13 Iterative Code Editor

The main workload for the code editor is navigating the code to find the particular code elements of interest for the edit. This is done with index management using the standard python functions `find()`, and `rfind()` to look for characteristics of the code elements. When considering addition of new states and events the prime objective is to retain the manually written code that might have been added to the code skeleton. When considering deletion of already existing states and events an additional important objective is to make sure that the trace of the element is removed in all the code elements of the original code skeleton. The characteristics vary for the different implementations and therefore the code editor is limited to one of the implementations and can be seen as a proof of concept. Because the Array of Structs implementation showed the best performance in the initial testing of the case study, it is the implementation that is considered.

3.13.1 Add State

Adding a state is the most simple action when editing the state machine. In the Array of Structs implementation the enumerate type containing the states, has an additional fictional "lastState" entry, the editor looks for the "lastState" and inserts the new state above.

3.13.2 Add Event

When adding an event, the editor has to add three code elements: a new event to the enumerate type, a new `eventHandler` function, and a new struct to the array. To add the event to the enumerate type, the editor looks for the "lastEvent" entry and inserts the new event above. To add the `eventHandler` function, the editor looks for the first code section after the `eventHandler` functions, in the Array of Structs implementation it is the "stateMachineStruct" array, the editor inserts the new `eventHandler` fucntion before the array. To add the new struct to the array, the editor looks for the last struct in the "stateMachineStruct" array by finding the last `eventHandler` function pointer in the array and inserting the new struct below.

3.13.3 Delete State

When deleting a state, the editor first removes the state from the enumerate type, secondly removes all `eventHandler` functions returning the state and the structs and events in the enumerate type related to those `eventHandler` functions, and finally removes all structs which includes the state and the `eventHandler` functions and events in the enumerate type related to those structs. When deleting elements from a specific code section, the editor first finds the specific code section and then looks for the element within that code section. To remove the state from the enumerate type, the editor simply looks for the state within the code section containing the states enumerate type. To remove the `eventHandler` functions returning the state, the editor looks for return statements followed by the state name within the `eventHandler` functions code section. The editor saves the names of the removed `eventHandler` functions and looks for them within the array of structs, if any are found, the editor removes the structs from the array and the events related to the structs from the enumerate type. To remove all structs which includes the state, the editor looks for the state name within the array of struct. If any structs are found, before removing them, the editor saves the event name from that struct and removes the event from the enumerate type and its associated `eventHandler` function.

3.13.4 Delete Event

When deleting an event, the editor finds and removes the event form the enumerate type, removes the struct containing the event, and removes the `eventHandler` function related to the event. The editor simply looks for the event name within the each particular code section.

3.14 Benchmarking

In the Python scripts that generate C-code using parsed data, there is also an option to generate test functions that does the benchmark. The execution time benchmark consists of generating a large amount of events and letting the state machine handle them. This is done 100 times and then averaged out to eliminate noise as much as possible. A total of 5 runs per measurement is done where the fastest time is selected as a data point. These events are required to be valid for the active state of the state machine, meaning that invalid events are discarded and regenerated until a valid one is found. This decision was made because the code skeletons does not include error handling, so upon receiving an invalid event, the state machine would immediately exit the event handler functions or do nothing which is not very interesting. Naturally, the timings only includes the time taken for the state machine to handle valid events. To monitor memory usage, the executable size is mainly used, along with the data distribution of the file using the `size` command in unix. Some memory monitoring tools like valgrind was used, but was not of much help due to the implementations not dynamically allocating memory for the most part. The stack allocated memory used was kept the same size during runtime.

For the scalability measurements, the scripts include functionality to generate random state machines. The number of valid events that the state machines handle are fixed, while the amount of states are changed. When generating state machines, there are some concerns to take into consideration to make measurements fair. For example, how do we generate the transition logic between states? Let us say a randomly generated state machine has every state having one state being the target and source of one transition each. That would be biased towards the Nested Switch implementation since the state would have a low amount of condition checking, meanwhile for another implementation like the state pattern using class methods, it would not make a difference. Therefore, a lower bound and upper bound on the amount of transitions are set up as a rule for the randomly generated state machines. The main rule is that all states should be the target and source of at least one transition. Then, the lower bound measurements has every state having one transition and the upper bound has each state being able to transition to every other state ($n^2 - n$ transitions). An additional measurement for state machines with completely random amounts of transitions is also considered as a kind of middle-ground. The measurements are done with both no optimisation flags and the -O2 optimisation flag to see if the compiler manages to optimise performance better for some implementations.

The interaction models measure scalability using a similar approach, but instead of measuring the best

case and worse case scenarios in terms of the amount of events in generated state machines, the best case and worse case scenarios consider the amount of actions that can be taken depending on the other state machine's active state. One could vary the amount of events along with the amount of actions as well, however for these measurements it was decided to only have one event for each state, as the generated code for a large amount of possible actions already created large code, with some of them not able to compile in gcc. A scalability measurement for interaction with 1000 states means both state machines have 1000 states. A run time measurement with 10^5 generated events for this case has each state machine running one event, then the other state machine running the next event, resulting in $5 \cdot 10^4$ events per state machine.

4 Results

4.1 Benchmarks

This section presents the benchmark results from measuring run time and memory size of the different implementations with both randomly generated state machines and the state machine from the case study. The benchmarks are all done on a Linux RedHat virtual machine, using an Intel Xeon Gold 6240R processor with 2.40 GHz. Measurements being specified as "-" in tables means the code was not successfully compiled. Ericsson does not specify exactly what type of optimisation is being used in their environments, but after discussion with the department we were told that the -O2 optimisation flag would make for a fairly similar optimisation. The run times presented here are the times it takes for the state machines to perform 10^5 transitions between states.

4.1.1 Nested Switch

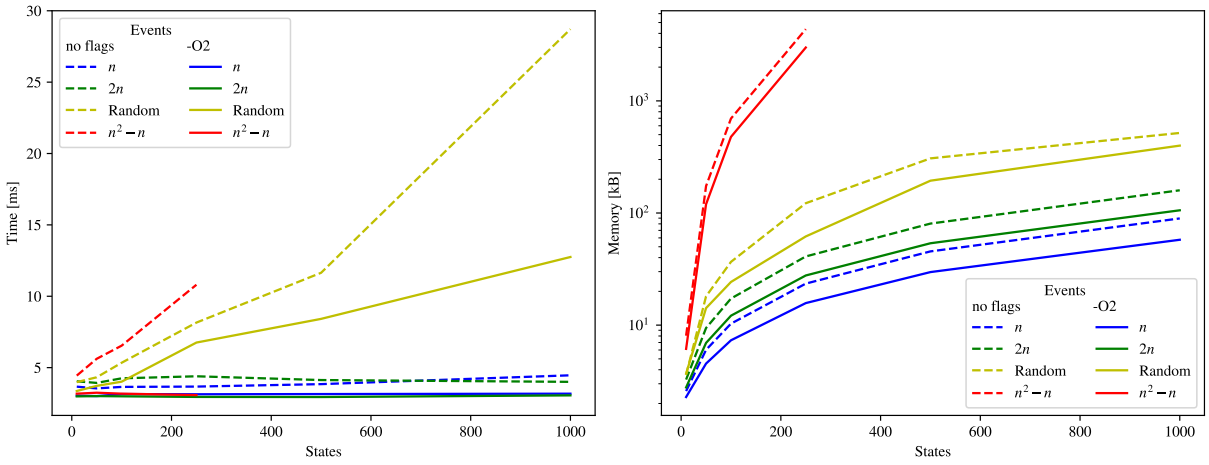
Tables 1 & 2, and Figure 13 show the results of the scalability benchmarks for the Nested Switch implementation. The results show more or less constant run time for the smaller state machines with n and $2n$ events, regardless of optimisation flags. When running without optimisation flags, the state machines with random and $n^2 - n$ events results in run times that increases with the size of the state machines, however the -O2 optimisation flag successfully reduces $n^2 - n$ events to constant run time. Overall, the -O2 optimisation flag reduce the run time by 14-72% (ignoring the random events). The size of the files increases with the size of the state machines, and the -O2 optimisation flag reduces the file sizes by 11-35% (ignoring the random events).

			Timings [ms]			
States [n]	Transitions	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	10^5	no flags	3.667	4.041	3.984	4.442
50	10^5	no flags	3.545	3.930	4.324	5.619
100	10^5	no flags	3.650	4.251	5.341	6.539
250	10^5	no flags	3.671	4.393	8.156	10.806
500	10^5	no flags	3.844	4.130	11.645	-
1000	10^5	no flags	4.462	4.003	28.713	-
10	10^5	-O2	3.039	2.986	3.362	3.175
50	10^5	-O2	3.005	3.006	3.722	3.242
100	10^5	-O2	3.148	2.991	4.016	3.176
250	10^5	-O2	3.134	2.947	6.757	3.076
500	10^5	-O2	3.148	2.940	8.416	-
1000	10^5	-O2	3.175	3.057	12.750	-

Table 1: Timings for the scalability benchmarks using the Nested Switch implementation.

		Memory [B]			
States [n]	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	no flags	2 580	3 268	3 612	8 052
50	no flags	6 020	9 428	18 068	173 892
100	no flags	10 308	17 268	36 668	696 292
250	no flags	23 460	40 980	122 268	4 363 492
500	no flags	45 460	80 484	307 084	17 475 492
1000	no flags	89 460	159 476	517 620	-
10	-O2	2 284	2 780	3 692	6 140
50	-O2	4 540	6 940	14 188	119 740
100	-O2	7 324	12 140	24 268	477 740
250	-O2	15 724	27 724	61 804	2 991 724
500	-O2	29 724	53 724	194 220	-
1000	-O2	57 724	105 724	399 052	-

Table 2: File sizes for the scalability benchmarks using the Nested Switch implementation.



(a) Time **(b)** Size

Figure 13: The Nested Switch implementations scalability benchmarks.

4.1.2 Function Pointers

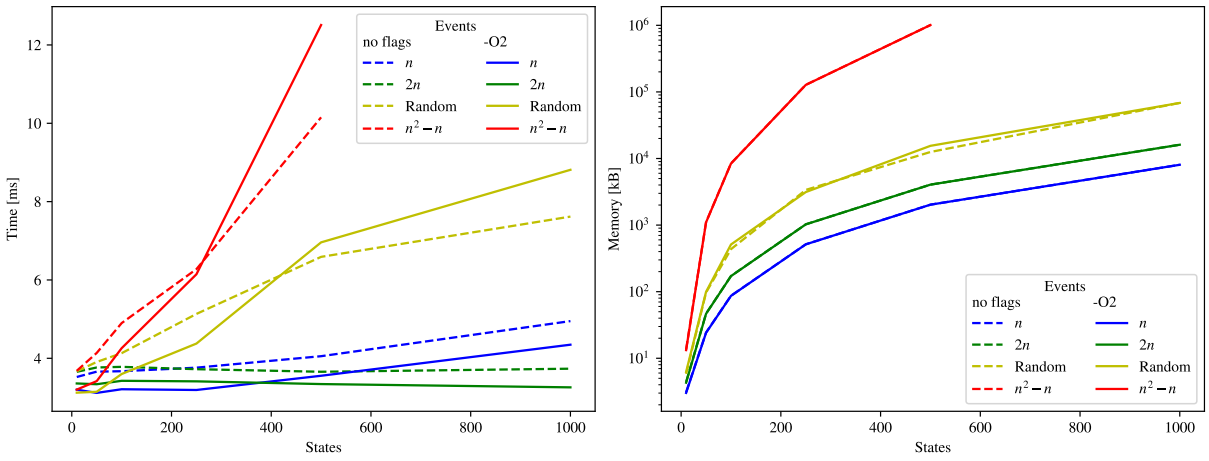
Tables 3 & 4, and Figure 14 displays the results from the scalability benchmarks of the Function Pointers implementation. The $2n$ events state machines manages a constant run time while the other state machines results in run times increasing with the size of the state machine. This means that the n events state machines has a longer run time than the $2n$ events state machines when the number of states surpasses ≈ 250 states. The results indicates that for the smaller state machines with n & $2n$ events, the -O2 optimisation flag consistently improves run time with 8-15%, while the results are more noisy and show no consistent improvement for the larger state machines. When it comes to memory, the file size simply increases with the size of the state machine, and the result is practically unaffected by the -O2 optimisation flag with a maximum memory reduction of 2.4% (ignoring the random events).

			Timings [ms]			
States [n]	Iterations	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	10^5	no flags	3.523	3.655	3.667	3.681
50	10^5	no flags	3.659	3.766	3.907	4.137
100	10^5	no flags	3.670	3.782	4.131	4.899
250	10^5	no flags	3.761	3.722	5.133	6.277
500	10^5	no flags	4.054	3.656	6.590	10.152
1000	10^5	no flags	4.950	3.735	7.618	-
10	10^5	-O2	3.198	3.358	3.122	3.201
50	10^5	-O2	3.117	3.338	3.145	3.417
100	10^5	-O2	3.210	3.425	3.598	4.258
250	10^5	-O2	3.191	3.412	4.376	6.146
500	10^5	-O2	3.554	3.342	6.960	12.509
1000	10^5	-O2	4.347	3.258	8.812	-

Table 3: Timings for the scalability benchmarks using the Function Pointers implementation.

		Memory [B]			
States [n]	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	no flags	3 092	4 404	5 836	13 556
50	no flags	24 340	46 884	96 940	1 106 724
100	no flags	86 884	171 988	434 940	8 426 676
250	no flags	514 516	1 027 268	3 375 676	127 676 516
500	no flags	2 027 268	4 052 772	12 531 516	1 010 726 276
1000	no flags	8 052 772	16 103 780	68 386 964	-
10	-O2	3 020	4 300	6 236	13 260
50	-O2	24 140	46 556	99 404	1 099 340
100	-O2	86 556	171 356	511 388	8 396 940
250	-O2	513 740	1 025 740	3 133 132	127 489 740
500	-O2	2 025 740	4 049 740	15 501 532	1 009 977 740
1000	-O2	8 049 740	16 097 740	68 329 260	-

Table 4: File sizes for the scalability benchmarks using the Array of Structs implementation.



(a) Time **(b)** Size

Figure 14: Function Pointers implementation scalability benchmarks.

4.1.3 Array Of Structs

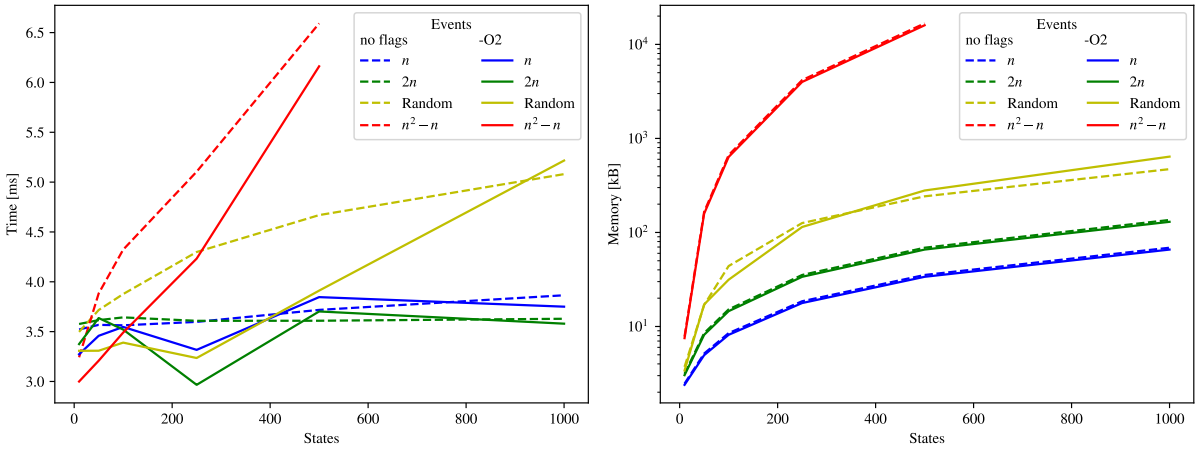
Tables 5 & 6, and Figure 15 displays the results from the scalability benchmarks of the Array of Structs implementation. When running the state machines without optimisation flags, the $2n$ events state machines keeps a consistent run time independent of the size of the state machines, while the run time increases with size for the other state machines. This means that the $2n$ events state machines runs faster than the smaller n events state machines when more than 250 states are considered. The results of using the -O2 optimisation flag is less conclusive with fluctuations that in some cases result in worse run times than without the -O2 flag. No matter the number of events, the -O2 flag results in particularly large reduction of run time for the state machines with exactly 250 states. The file size increase with the size of the state machines, and the -O2 optimisation flag reduces the file sizes by 2.9-4.5% (ignoring the random events).

			Timings [ms]			
States [n]	Iterations	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	10^5	no flags	3.522	3.577	3.498	3.243
50	10^5	no flags	3.568	3.615	3.717	3.883
100	10^5	no flags	3.563	3.643	3.877	4.323
250	10^5	no flags	3.598	3.608	4.299	5.105
500	10^5	no flags	3.718	3.609	4.669	6.592
1000	10^5	no flags	3.865	3.629	5.080	-
10	10^5	-O2	3.273	3.375	3.308	2.999
50	10^5	-O2	3.458	3.636	3.308	3.208
100	10^5	-O2	3.547	3.521	3.390	3.490
250	10^5	-O2	3.317	2.966	3.235	4.232
500	10^5	-O2	3.846	3.702	3.911	6.162
1000	10^5	-O2	3.750	3.580	5.216	-

Table 5: Timings for the scalability benchmarks using the Array of Structs implementation.

		Memory [B]			
States [n]	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	no flags	2 468	3 140	3 724	7 828
50	no flags	5 156	8 500	16 932	165 956
100	no flags	8 500	15 204	43 932	665 108
250	no flags	18 548	35 300	125 860	4 172 548
500	no flags	35 300	68 804	242 180	16 718 308
1000	no flags	68 804	135 812	471 316	-
10	-O2	2 396	3 036	3 404	7 516
50	-O2	4 956	8 156	17 164	158 556
100	-O2	8 156	14 556	31 436	635 356
250	-O2	17 756	33 756	114 316	3 985 756
500	-O2	33 756	65 756	279 308	15 969 756
1000	-O2	65 756	129 756	640 268	-

Table 6: File sizes for the scalability benchmarks using the Array of Structs implementation.



(a) Time **(b)** Size

Figure 15: Array of Structs implementation scalability benchmarks.

4.1.4 Basic State Pattern

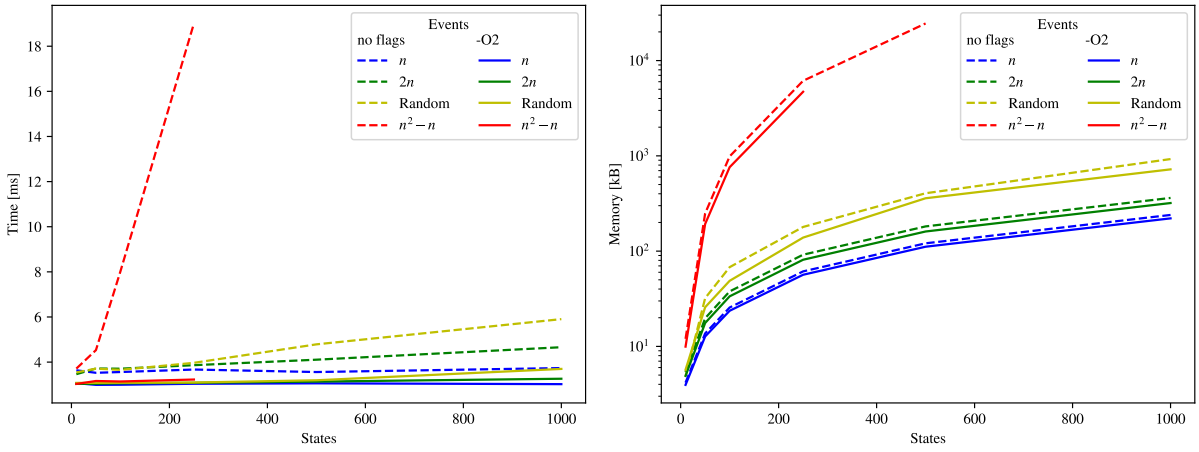
Tables 7 and 8 show the results of the scalability measurements for the state pattern implementation, visualized in Figure 16. While a bit noisy, the best case scenario results for a generated state machine with n events seems to indicate that the implementation scales well, in both the cases of optimisation. For generated state machines with $2n$ and random amounts of events there is an increase in run time for increasing number of states, which means the implementation does not scale as well in those cases. The worst case scenario with $n^2 - n$ events seem to not scale well at all, but when using optimisation flags the opposite could be the case looking at the timings. However, the reduced amount of samples makes it hard to discern since there is no data for 500 and 1000 states.
Memory size for the State Pattern is shown to be high, in particular when increasing the amount of events. A generated state machine with $n^2 - n$ events and 500 states compiled with no flags, have an executable size of about 24.5 MB for example. With optimisation the compiler manages to reduce the memory size by a noticeable amount.

			Timings [ms]			
States [n]	Iterations	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	10^5	no flags	3.637	3.465	3.542	3.712
50	10^5	no flags	3.526	3.716	3.710	4.531
100	10^5	no flags	3.561	3.708	3.657	8.023
250	10^5	no flags	3.671	3.864	3.968	19.009
500	10^5	no flags	3.560	4.107	4.787	-
1000	10^5	no flags	3.736	4.660	5.905	-
10	10^5	-O2	3.067	3.038	3.078	3.032
50	10^5	-O2	2.991	3.039	3.081	3.163
100	10^5	-O2	2.997	3.060	3.083	3.141
250	10^5	-O2	3.040	3.081	3.095	3.228
500	10^5	-O2	3.060	3.136	3.195	-
1000	10^5	-O2	3.023	3.263	3.698	-

Table 7: Timings for the scalability benchmarks using the State Pattern implementation.

		Memory [B]			
States [n]	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	no flags	4 171	5 363	4 883	12 011
50	no flags	13 691	19 763	32 107	248 891
100	no flags	25 595	37 779	67 883	987 587
250	no flags	61 291	91 763	179 403	6 137 299
500	no flags	120 795	181 779	405 067	24 530 787
1000	no flags	239 803	361 779	925 403	-
10	-O2	3 947	4 899	5 691	9 947
50	-O2	12 715	17 619	25 771	192 715
100	-O2	23 659	33 539	48 803	760 251
250	-O2	56 507	81 219	138 747	4 706 507
500	-O2	111 259	160 739	358 931	-
1000	-O2	220 763	319 731	721 563	-

Table 8: File sizes for the scalability benchmarks using the State Pattern implementation.



(a) Run time

(b) Memory size

Figure 16: State Pattern implementation scalability benchmarks.

4.1.5 State-Table Pattern

Results for the State-Table Pattern approach is given in Tables 9 and 10, along with a plot in Figure 17. Here, the run time increases for an increasing amount of states for all cases of events in the generated state machines, suggesting that it does not scale well in terms of performance using this implementation. The only case for when the run time is more consistent is for n events, using the -O2 optimisation flag when compiling.

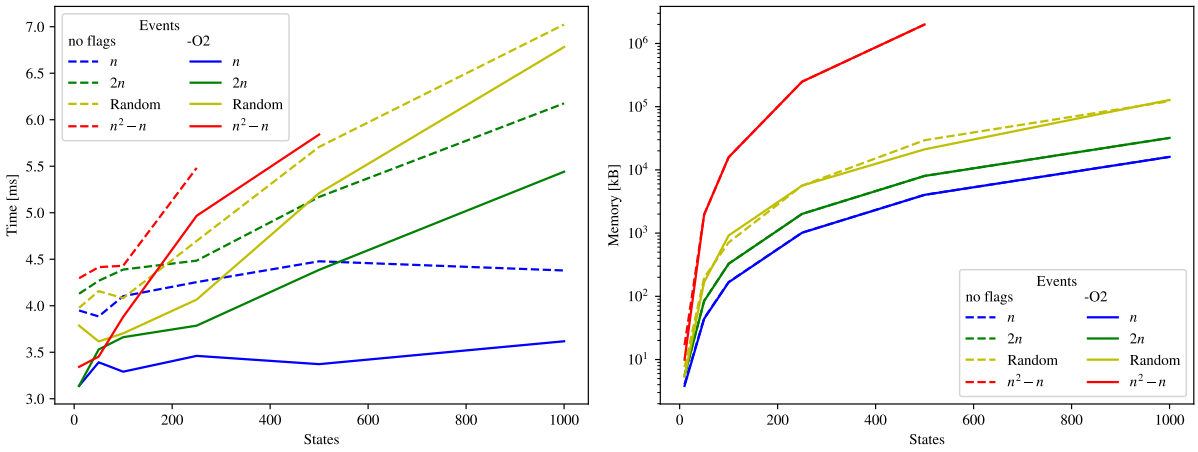
The measurements show that the State-Table Pattern uses up most amount of memory of any of the implementations tested. Increasing the number of states increases the lookup table used which requires more allocation memory, as well as the array storing `EventTable` structs containing more than just a normal function pointer. Using optimisation flags, the memory size is reduced but by a small proportion as for 500 states with $n^2 - n$ events for instance, the memory size is reduced by less than $10^{-5}\%$.

			Timings [ms]			
States [n]	Iterations	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	10^5	no flags	3.949	4.129	3.978	4.296
50	10^5	no flags	3.885	4.268	4.157	4.415
100	10^5	no flags	4.103	4.389	4.082	4.430
250	10^5	no flags	4.254	4.485	4.697	5.482
500	10^5	no flags	4.478	5.170	5.708	-
1000	10^5	no flags	4.379	6.176	7.024	-
10	10^5	-O2	3.138	3.140	3.786	3.343
50	10^5	-O2	3.392	3.530	3.616	3.453
100	10^5	-O2	3.291	3.662	3.703	3.880
250	10^5	-O2	3.461	3.786	4.066	4.967
500	10^5	-O2	3.372	4.386	5.211	5.841
1000	10^5	-O2	3.618	5.441	6.782	-

Table 9: Timings for the scalability benchmarks using the State-Table Pattern implementation.

		Memory [B]			
States [n]	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	no flags	4 063	5 663	7 583	16 863
50	no flags	44 991	85 007	196 991	1 964 975
100	no flags	168 143	328 143	710 527	15 848 127
250	no flags	1 017 583	2 017 583	5 637 583	249 017 583
500	no flags	4 033 327	8 033 327	29 449 327	1 996 033 327
1000	no flags	16 064 831	32 064 831	122 176 831	-
10	-O2	3 823	5 423	5 423	9 947
50	-O2	44 143	84 143	167 343	1 964 143
100	-O2	166 543	326 543	915 343	15 846 543
250	-O2	1 013 743	2 013 743	5 645 743	249 013 743
500	-O2	4 025 743	8 025 743	21 137 743	1 996 025 743
1000	-O2	16 049 743	32 049 743	127 873 743	-

Table 10: File sizes for the scalability benchmarks using the State-Table Pattern implementation.



(a) Run time

(b) Memory size

Figure 17: State-Table Pattern implementation scalability benchmarks.

4.1.6 Hierarchical State Pattern

Looking at Tables 11, 12 and Figure 18, the results for the Hierarchical State Pattern are shown. From the measurements, it is implied that generated state machines of all cases increase in run time when the state machine grows. For state machines generated with $n^2 - n$ events, run time measured is significantly higher compared to the best case scenario of n events. With optimisation, run time is reduced by a noticeable margin and close to constant for n events.

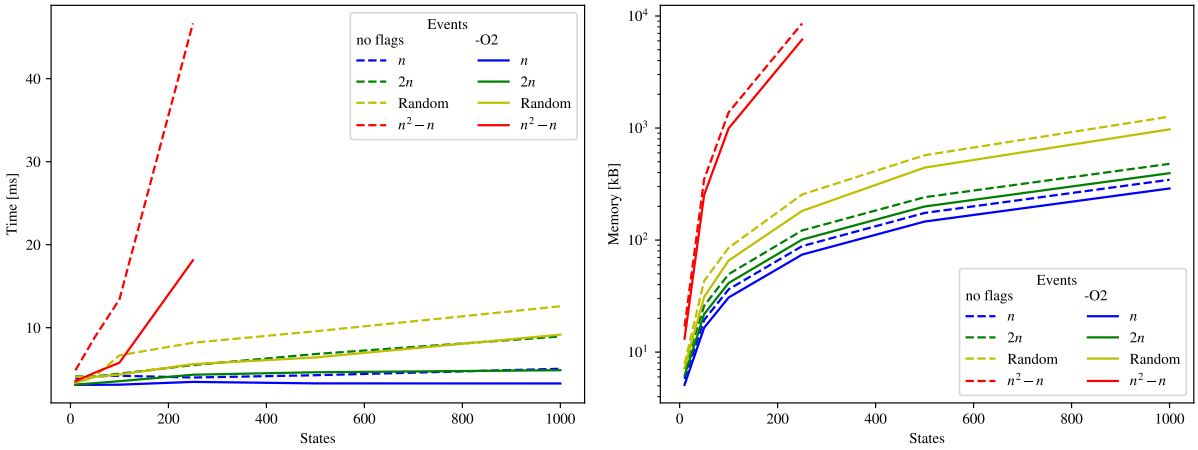
The added functionality for hierarchy, history and concurrency makes the implementation need more memory than the regular state pattern approach. Including the -O2 optimisation flag when compiling reduces this size further.

			Timings [ms]			
States [n]	Iterations	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	10^5	no flags	3.841	4.129	4.062	4.851
50	10^5	no flags	4.140	4.136	4.125	8.888
100	10^5	no flags	4.212	4.463	6.669	13.404
250	10^5	no flags	3.998	5.494	8.189	46.661
500	10^5	no flags	4.273	6.825	9.567	-
1000	10^5	no flags	5.052	8.948	12.577	-
10	10^5	-O2	3.131	3.136	3.357	3.542
50	10^5	-O2	3.117	3.322	4.082	4.590
100	10^5	-O2	3.137	3.555	4.349	5.781
250	10^5	-O2	3.470	4.334	5.610	18.121
500	10^5	-O2	3.291	4.629	6.415	-
1000	10^5	-O2	3.281	4.879	9.166	-

Table 11: Hierarchical State Pattern implementation scalability benchmarks.

		Memory [B]			
States [n]	Flags	n Events	$2n$ Events	Random Events (min $2n$)	$n^2 - n$ Events
10	no flags	5 759	7 047	7 855	16 887
50	no flags	19 351	25 911	43 231	349 367
100	no flags	36 311	49 607	85 255	1 381 487
250	no flags	88 279	121 815	255 263	8 588 063
500	no flags	174 551	241 175	571 447	-
1000	no flags	345 559	479 287	1 263 967	-
10	-O2	5 103	6 151	7 279	13175
50	-O2	16 503	21 815	31 127	253127
100	-O2	30 727	41 367	65639	994087
250	-O2	74 183	100 967	182 103	6156711
500	-O2	146 215	199 335	443 367	-
1000	-O2	288 727	395 447	972 967	-

Table 12: File sizes for the scalability benchmarks using the Hierarchical State Pattern implementation.

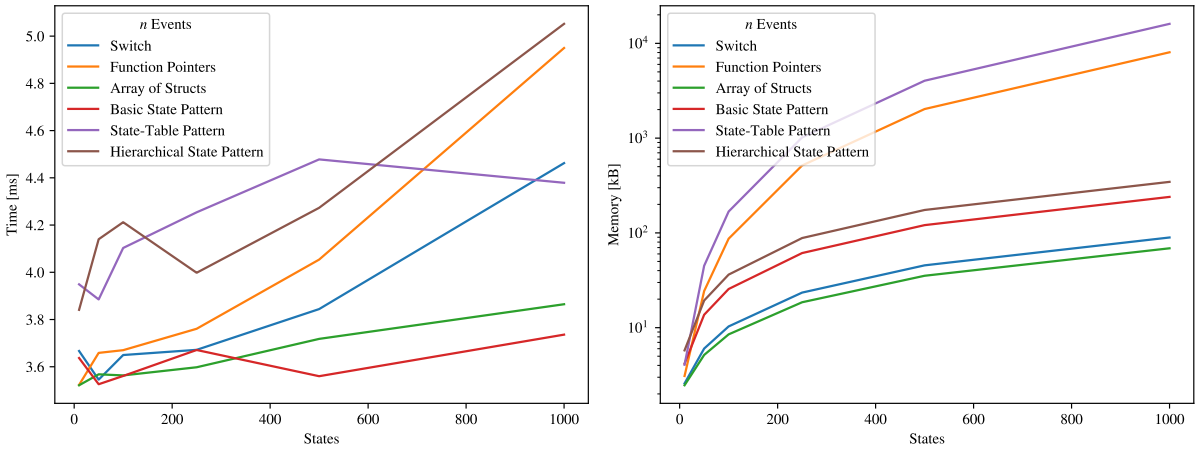


(a) Run time **(b)** Memory size

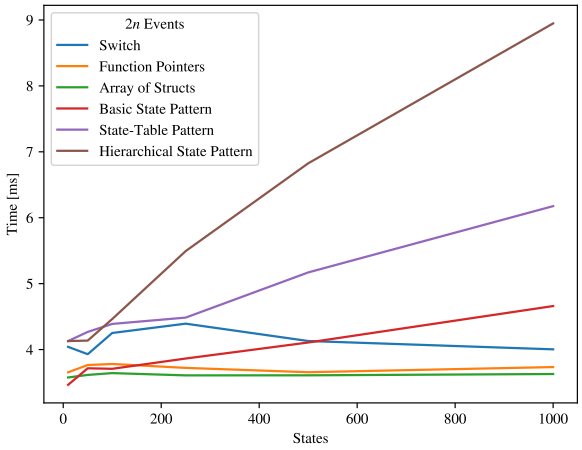
Figure 18: Run time and memory size measurements for the Hierarchical State Pattern.

4.2 Comparing Benchmarks

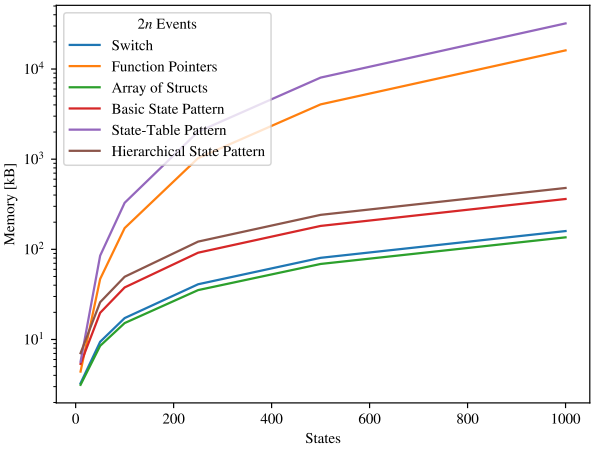
Looking at all implementations at the same time in Figure 19 with no optimisation flags, for the cases of state machines generated with n, $2n$ and random events the State Pattern and Array of Structs approach performs the best in terms of run time. For any case of event size, the Hierarchical State Pattern scales the worst and is the slowest barring the cases when the amount of events are random or $2n$ in the generated state machine, where the Nested Switch implementation is slower. The 2D array based State-Table Pattern and Function Pointers solutions rank in the middle of the approaches and perform well, similar to the State Pattern and Array of Structs for event sizes bigger than n. Using the Nested Switch implementation does not seem to work the best for state machines with $2n$ and random events, according to the results. However, the non pattern-based Nested Switch and Array of Structs ideas depend on memory the least out of the implementations, while the extra structure overhead of the pattern based implementations are in need of more. Requiring the most is the State-Table pattern and Function Pointers that use a 2D-array to handle events.



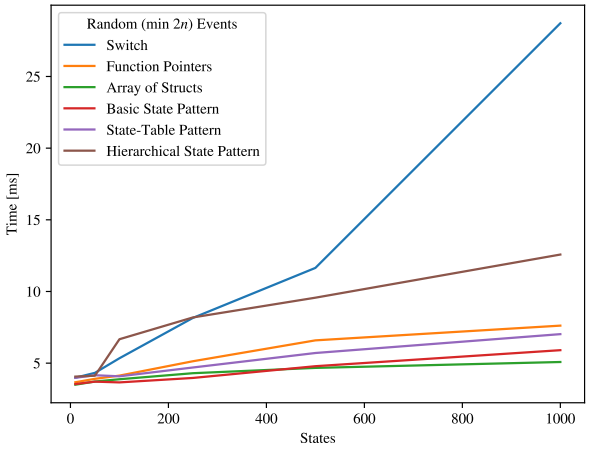
(a) Time n **(b)** Size n



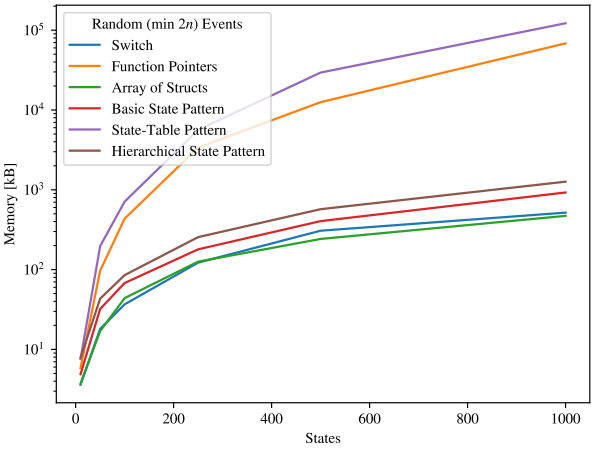
(c) Time $2n$



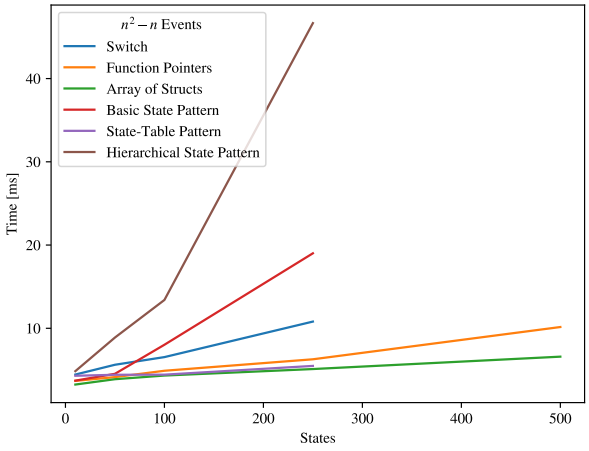
(d) Size $2n$



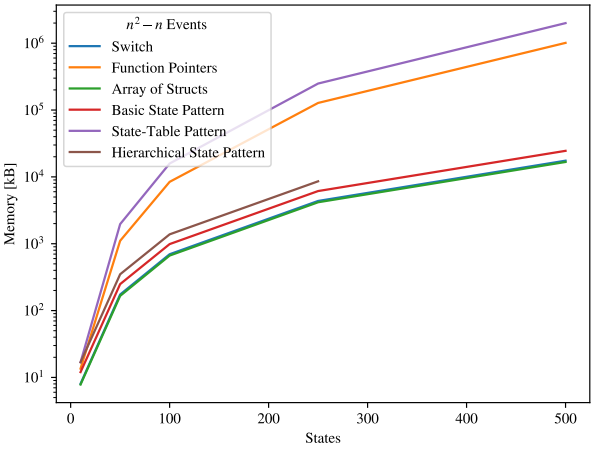
(e) Time Random (min $2n$)



(f) Size Random (min $2n$)



(g) Time $n^2 - n$

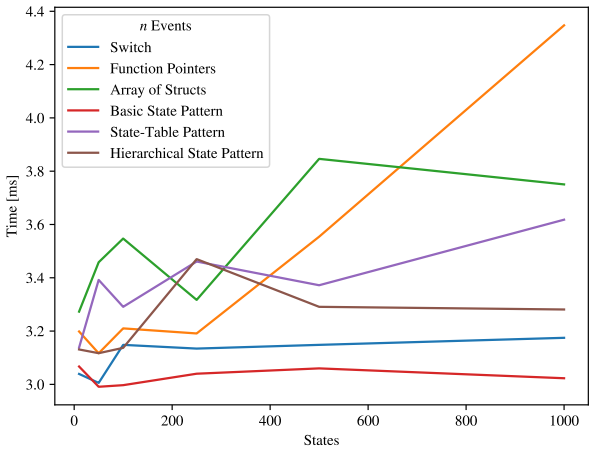


(h) Size $n^2 - n$

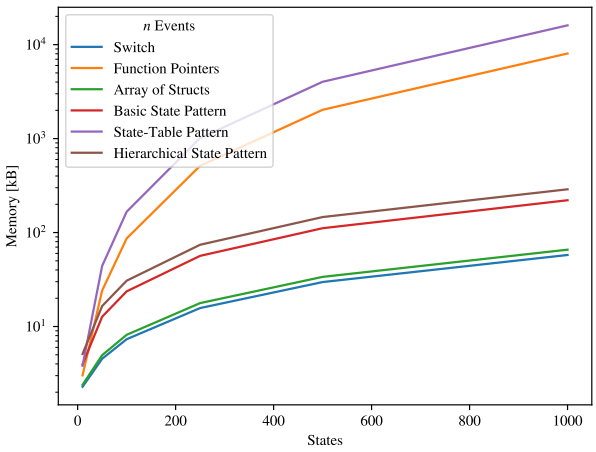
Figure 19: Run time and memory size for state machine implementations, no optimisation flags.

With the -O2 optimisation flag added in Figure 20, we notice that the State Pattern implementation performs well in all cases of generated state machines, and the run time for the n^2-n case has been greatly

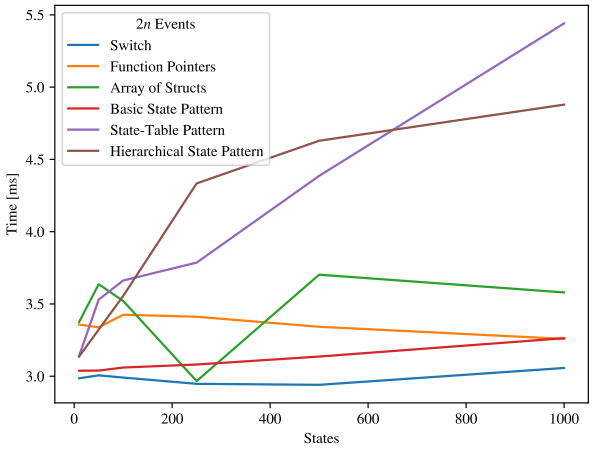
reduced in the optimisation by the compiler. The Array of Structs implementation still performs well but a difference from the benchmark with no optimisation flags is that the Nested Switch implementation seem to have improved, competing with the State Pattern approach except for state machines generated with random events. Both table based approaches State-Table Pattern and Function Pointers compare worse to most other implementations, especially for larger state machines with more states and events. For smaller amounts of events, the added structure of the Hierarchical State Pattern does not seem to impact the performance too much, but for $n^2 - n$ its run time is the worst. On the memory side, the table based State-Table Pattern and Function Pointers approaches needs the most memory for all cases of events. The Nested Switch and Array of Structs implementations have the smallest memory size, with the extra overhead from the pattern based State Pattern and Hierarchical State Pattern require more. The difference between these two and the Array of Structs and Nested Switch approaches is smaller the more events the state machines have however, since the extra structure needed for the State Pattern then matters less.



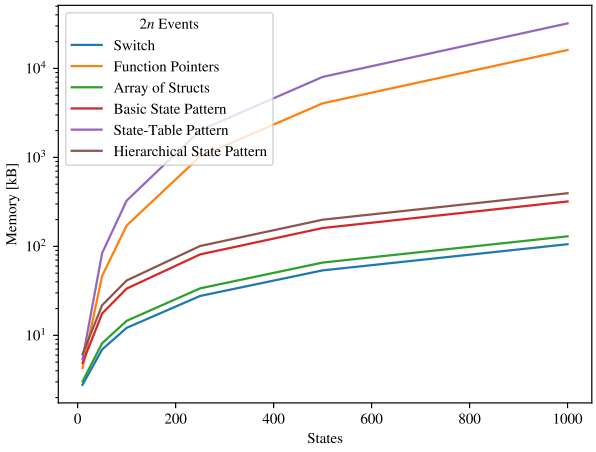
(a) Time n



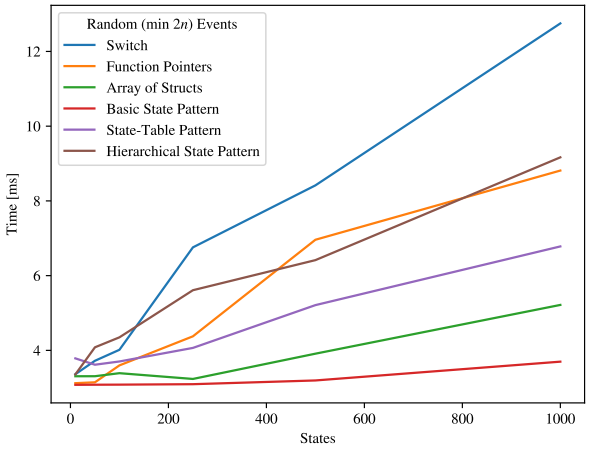
(b) Size n



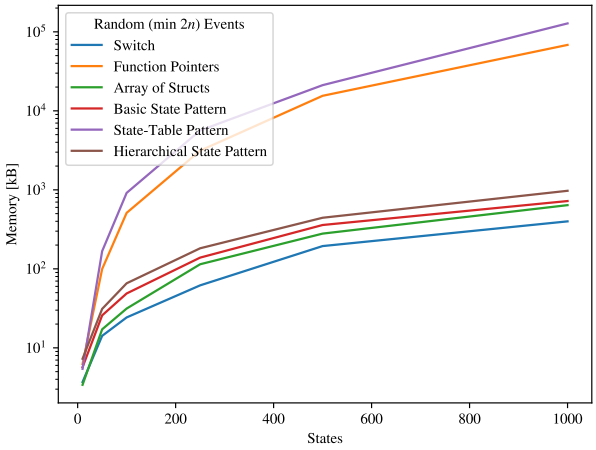
(c) Time $2n$



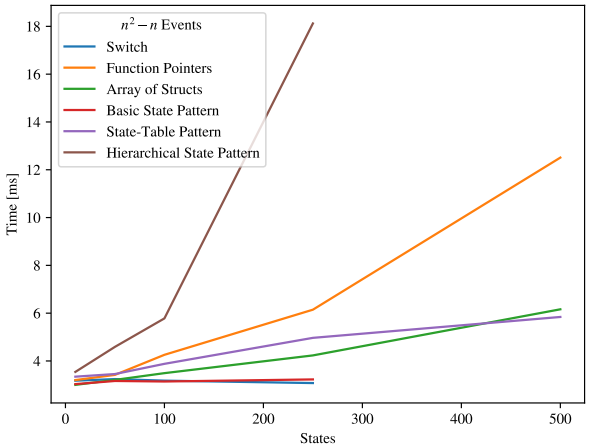
(d) Size $2n$



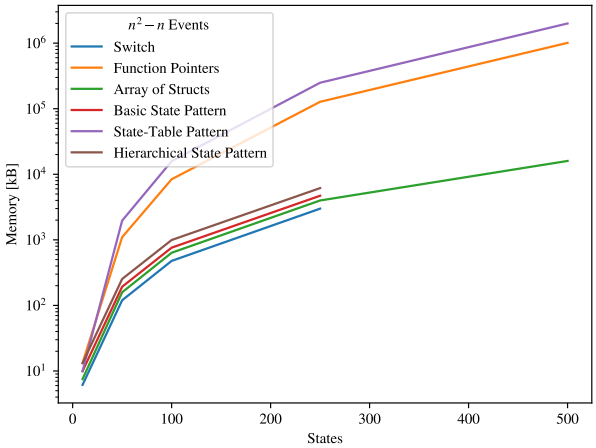
(e) Time Random (min $2n$)



(f) Size Random (min $2n$)



(g) Time $n^2 - n$



(h) Size $n^2 - n$

Figure 20: Run time and memory size for state machine implementations, optimised with -O2.

4.3 Case Study

The results for the case study are illustrated without optimisation flags in Figure 21, and with the -O2 optimisation flag in Figure 22. The results without optimisation flags suggest that the Array of Structs method both performs best in regards to run time, and requires the least amount of memory. Most of the state pattern based implementations ranks at the bottom, with the Hierarchical State Pattern having the longest run time and largest memory size. Using the Nested Switch approach, it is seen that the memory size is low, but slow for this system in regards to run time, possibly due to the case study having high amounts of transitions compared to states. When using the -O2 optimisation flag the Array of Structs implementation just about loses out to the Basic State Pattern implementation for run time and the Nested Switch implementation for memory usage.

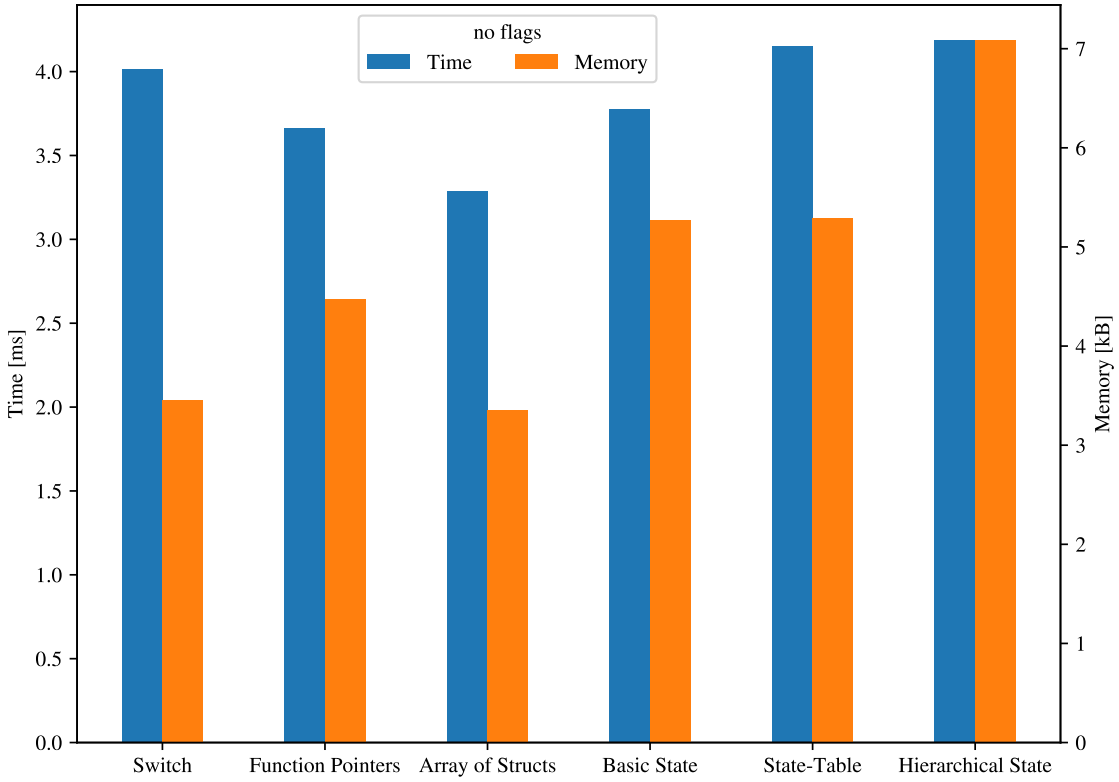


Figure 21: Result from the case study compiled without optimisation flags.

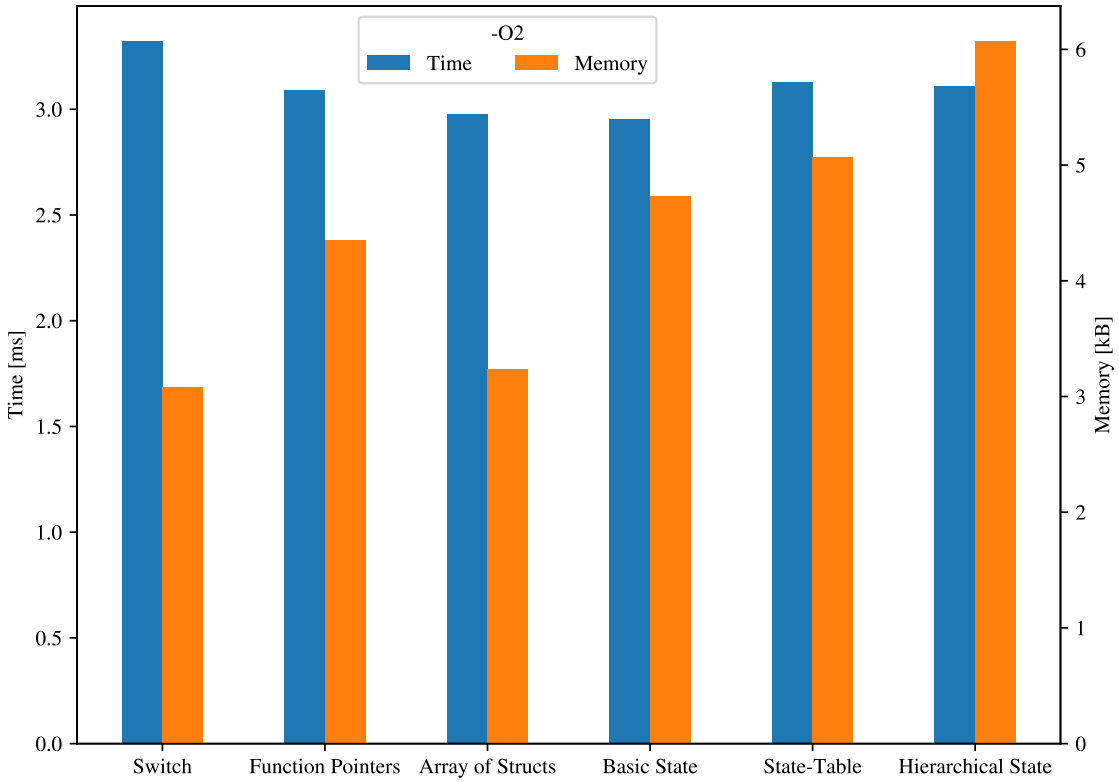


Figure 22: Result from the case study compiled with the -O2 optimisation flag.

4.4 State Machine Interaction Benchmarks

4.4.1 Global Switch

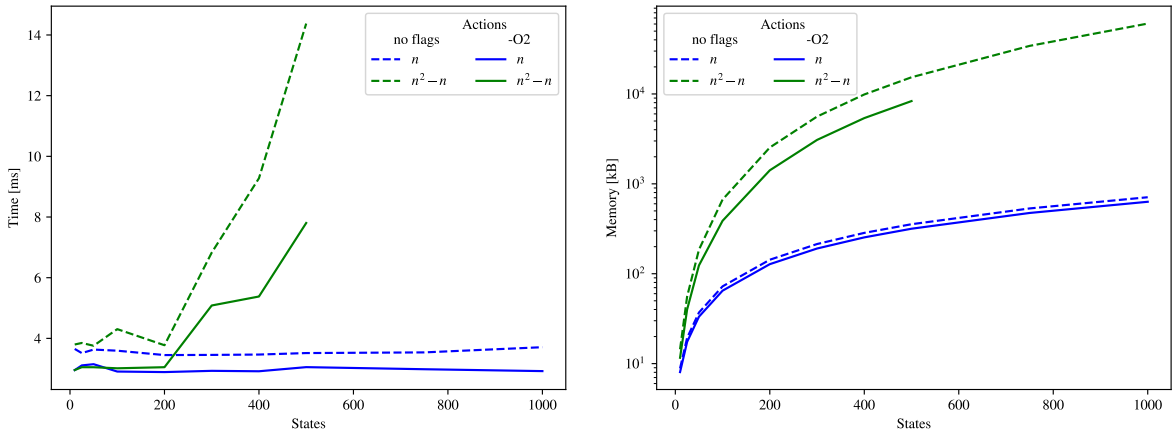
In Tables 13 & 14 and Figure 23, the run time and memory measurements are displayed using the Global Switch method. For the best case scenario of n actions the timings are consistent and seem to scale well for both the optimised and unoptimised case, with small variations from noise. For $n^2 - n$ actions the run time is high for measurements with 300+ states, which is dampened when using optimisation.

			Timings [ms]	
States [n]	Iterations	Flags	n Actions	$n^2 - n$ Actions
10	10^5	no flags	3.652	3.798
25	10^5	no flags	3.515	3.849
50	10^5	no flags	3.631	3.757
100	10^5	no flags	3.594	4.305
200	10^5	no flags	3.450	3.774
300	10^5	no flags	3.455	6.826
400	10^5	no flags	3.467	9.278
500	10^5	no flags	3.516	14.376
750	10^5	no flags	3.538	-
1000	10^5	no flags	3.709	-
10	10^5	-O2	2.954	2.956
25	10^5	-O2	3.109	3.050
50	10^5	-O2	3.149	3.049
100	10^5	-O2	2.907	3.014
200	10^5	-O2	2.889	3.050
300	10^5	-O2	2.930	5.084
400	10^5	-O2	2.916	5.380
500	10^5	-O2	3.052	7.803
750	10^5	-O2	2.983	-
1000	10^5	-O2	2.921	-

Table 13: Run time using switch statements in the event handlers with global variables.

		Memory [B]	
States [n]	Flags	n Actions	$n^2 - n$ Actions
10	no flags	8 851	14 451
25	no flags	19 379	55 891
50	no flags	37 011	185 059)
100	no flags	72 355	668 307
200	no flags	143 123	2 536 003
300	no flags	213 971	5 603 107
400	no flags	284 771	9 870 211
500	no flags	355 523	15 337 299
750	no flags	532 563	34 255 059
1000	no flags	709 539	60 672 803
10	-O2	8 051	11 619
25	-O2	17 475	40 115
50	-O2	33 267	123 731
100	-O2	64 771	387 971
200	-O2	127 763	1 414 163
300	-O2	190 771	3 080 371
400	-O2	253 763	5 386 563
500	-O2	316 771	8 332 771
750	-O2	474 275	-
1000	-O2	631 763	-

Table 14: Executable size using switch statements in event handlers with global variables.



(a) Time **(b)** Size

Figure 23: Performance measurements for interaction using switch statements in the event handlers with global state machine variables.

4.4.2 Mediator Pattern

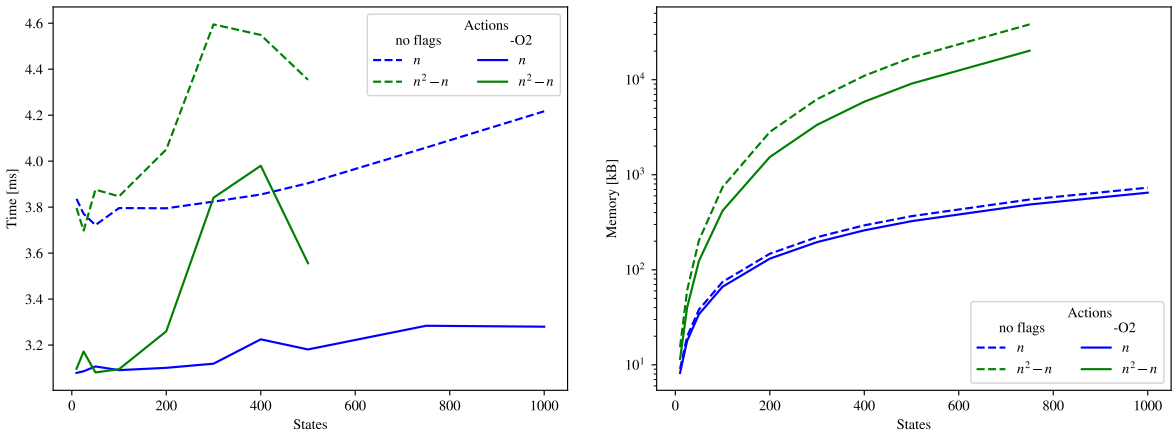
Using a mediator as communication we obtain the results given in Tables 15 & 16 and Figure 24. With n actions, the run time increases slightly for growing amounts of states. The timings for $n^2 - n$ shows an increase for more states, but then erratically decreases for 500 states. There is likely more noise in these measurements. The optimised timings point to a significant improvement for both the n and $n^2 - n$ cases, as well as slight scaling improvements. For n actions with optimisation the timings are close to constant. Memory size is reduced greatly as well.

			Timings [ms]	
States [n]	Iterations	Flags	n Actions	$n^2 - n$ Actions
10	10^5	no flags	3.836	3.796
25	10^5	no flags	3.771	3.698
50	10^5	no flags	3.722	3.876
100	10^5	no flags	3.796	3.847
200	10^5	no flags	3.795	4.051
300	10^5	no flags	3.824	4.595
400	10^5	no flags	3.855	4.549
500	10^5	no flags	3.904	4.354
750	10^5	no flags	4.059	-
1000	10^5	no flags	4.217	-
10	10^5	-O2	3.079	3.097
25	10^5	-O2	3.086	3.172
50	10^5	-O2	3.107	3.081
100	10^5	-O2	3.091	3.095
200	10^5	-O2	3.101	3.260
300	10^5	-O2	3.119	3.841
400	10^5	-O2	3.225	3.980
500	10^5	-O2	3.181	3.556
750	10^5	-O2	3.284	-
1000	10^5	-O2	3.280	-

Table 15: Run time using the mediator pattern.

		Memory [B]	
States [n]	Flags	n Actions	$n^2 - n$ Actions
10	no flags	9 059	15 363
25	no flags	19 923	60 803
50	no flags	38 099	203 603
100	no flags	74 531	740 403
200	no flags	147 539	2 820 211
300	no flags	220 547	6 239 411
400	no flags	293 523	10 998 611
500	no flags	366 579	17 097 811
750	no flags	549 027	38 208 307
1000	no flags	731 523	-
10	-O2	8 227	11 523
25	-O2	17 891	40 163
50	-O2	34 067	123 987
100	-O2	66 371	418 371
200	-O2	130 963	1 535 763
300	-O2	195 571	3 349 171
400	-O2	260 163	5 866 563
500	-O2	324 771	9 084 771
750	-O2	486 275	20 190 275
1000	-O2	647 763	-

Table 16: Executable size using the mediator pattern.



(a) Time

(b) Size

Figure 24: Performance measurements for interaction using the mediator pattern.

4.4.3 Message Passing

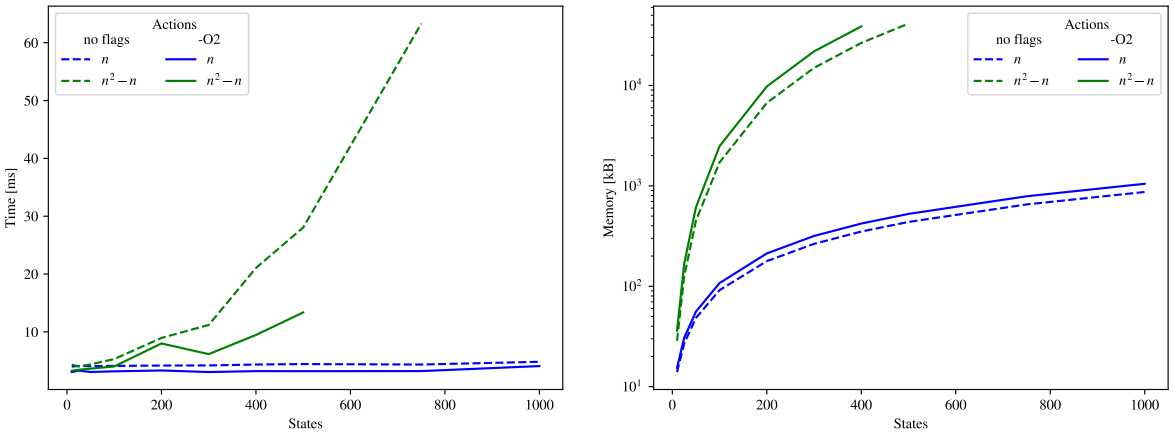
Results for adding message passing can be seen in Tables 17 & 18 and Figure 25. With both no optimisation and optimisation for n actions, the timings are fairly consistent until 1000 states. The timings for $n^2 - n$ actions with no flags record the highest timings for any of the interaction methods. With the -O2 optimisation flag, the run time is improved for all cases, but most noticably for $n^2 - n$ actions. Adding the message structs and message queue makes the implementation use more memory than for the Global Switch and Mediator Pattern methods. With optimisation the memory size instead increases.

			Timings [ms]	
States [n]	Iterations	Flags	n Actions	$n^2 - n$ Actions
10	10^5	no flags	4.002	4.340
25	10^5	no flags	4.116	4.015
50	10^5	no flags	4.076	4.389
100	10^5	no flags	4.119	5.304
200	10^5	no flags	4.182	8.955
300	10^5	no flags	4.201	11.202
400	10^5	no flags	4.357	21.052
500	10^5	no flags	4.436	28.044
750	10^5	no flags	4.357	63.307
1000	10^5	no flags	4.831	-
10	10^5	-O2	3.051	3.279
25	10^5	-O2	3.301	3.456
50	10^5	-O2	3.068	3.654
100	10^5	-O2	3.180	4.027
200	10^5	-O2	3.330	8.001
300	10^5	-O2	3.058	6.155
400	10^5	-O2	3.217	9.471
500	10^5	-O2	3.208	13.364
750	10^5	-O2	3.223	-
1000	10^5	-O2	4.089	-

Table 17: Run time using message passing.

		Memory [B]	
States [n]	Flags	n Actions	$n^2 - n$ Actions
10	no flags	13 971	28 627
25	no flags	26 835	125 011
50	no flags	48 371	449 731
100	no flags	91 491	1 714 083
200	no flags	177 939	6 703 971
300	no flags	264 339	14 973 283
400	no flags	350 723	26 522 579
500	no flags	437 155	41 351 875
750	no flags	653 091	-
1000	no flags	869 091	-
10	-O2	15 411	36 083
25	-O2	30 563	166 435
50	-O2	55 923	612 115
100	-O2	107 651	2 487 427
200	-O2	212 323	9 772 819
300	-O2	316 899	21 856 627
400	-O2	421 251	38 740 419
500	-O2	525 939	-
750	-O2	787 587	-
1000	-O2	1 049 155	-

Table 18: Executable size using message passing.



(a) Time **(b)** Size

Figure 25: Performance measurements for interaction using message passing.

4.4.4 Function Pointer 3D array

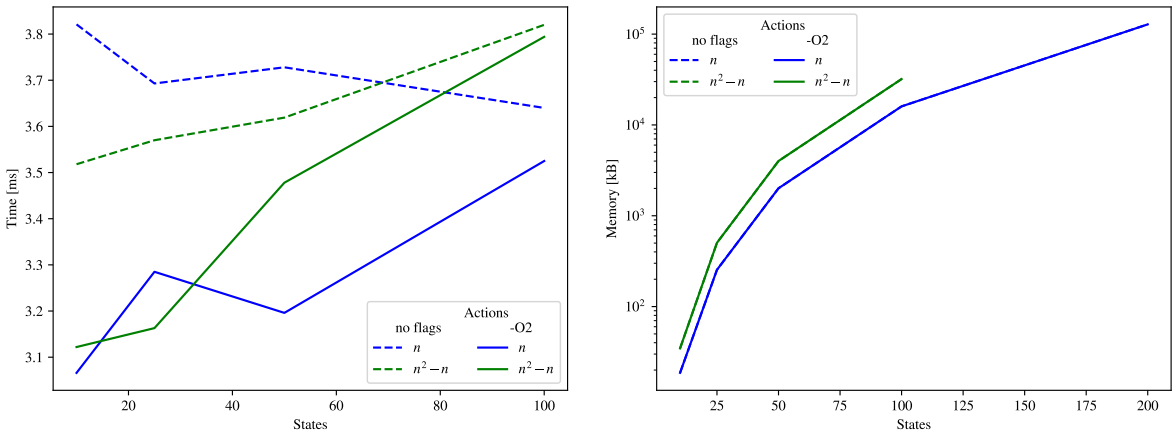
Results for the 3D array approach is given in Tables 19 & 20 and Figure 26. This method was not able to compile for two generated interacting state machines with states more than 100 states. This was due to running out of memory when the array was stack allocated. When trying to run for a heap allocated array instead, the code was too long from initializing the non-empty values which also made the code not compile. Nonetheless, it is seen that the difference in memory required between the best and worst case scenario is low, close to a factor of 2. The number of data points from the run time measurements in are not enough, and thus makes it not possible to see any real result.

			Timings [ms]	
States per machine [n]	Iterations	Flags	n Actions	$n^2 - n$ Actions
10	10^5	no flags	3.821	3.518
25	10^5	no flags	3.693	3.570
50	10^5	no flags	3.728	3.619
100	10^5	no flags	3.640	3.820
200	10^5	no flags	-	-
300	10^5	no flags	-	-
400	10^5	no flags	-	-
500	10^5	no flags	-	-
750	10^5	no flags	-	-
1000	10^5	no flags	-	-
10	10^5	-O2	3.066	3.122
25	10^5	-O2	3.285	3.163
50	10^5	-O2	3.196	3.478
100	10^5	-O2	3.525	3.794
200	10^5	-O2	-	-
300	10^5	-O2	-	-
400	10^5	-O2	-	-
500	10^5	-O2	-	-
750	10^5	-O2	-	-
1000	10^5	-O2	-	-

Table 19: Run time using 3D arrays.

		Memory [B]	
States [n]	Flags	n Actions	$n^2 - n$ Actions
10	no flags	18 615	34 615
25	no flags	254 031	504 023
50	no flags	2 006 295	4 006 295
100	no flags	16 010 903	32 010 903
200	no flags	128 020 103	-
300	no flags	-	-
400	no flags	-	-
500	no flags	-	-
750	no flags	-	-
1000	no flags	-	-
10	-O2	18 639	34 639
25	-O2	254 103	504 095
50	-O2	2 006 479	4 006 479
100	-O2	16 011 279	32 011 279
200	-O2	128 020 879	-
300	-O2	-	-
400	-O2	-	-
500	-O2	-	-
750	-O2	-	-
1000	-O2	-	-

Table 20: Executable size using 3D arrays.

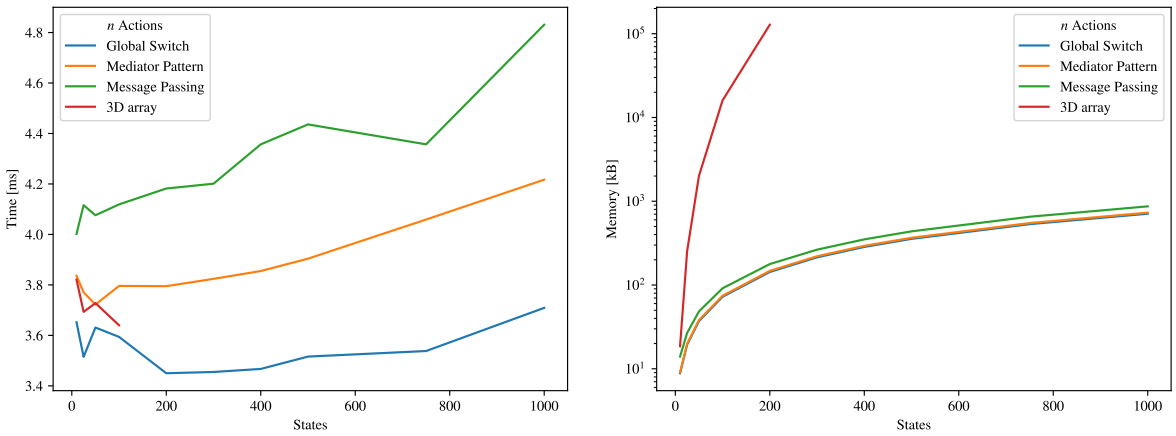


(a) Time **(b)** Size

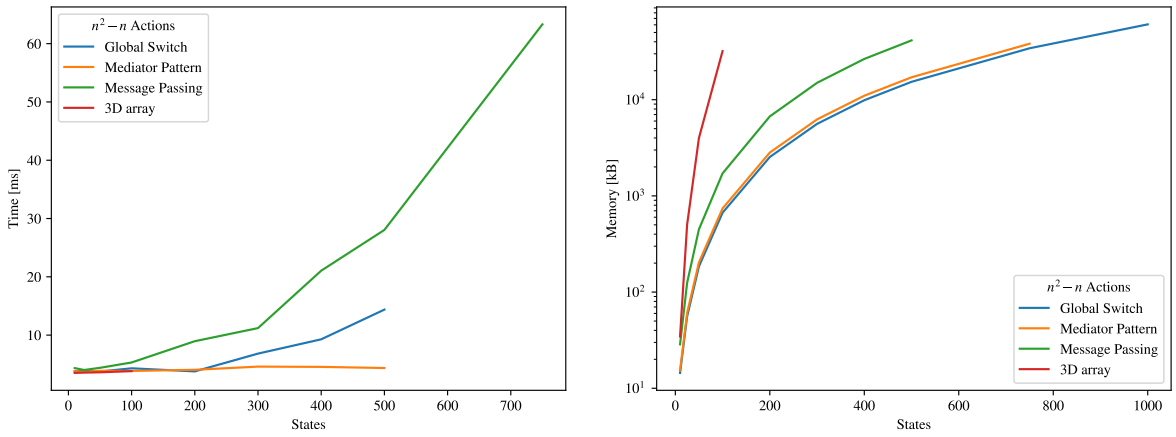
Figure 26: Performance measurements for interaction using 3D arrays.

4.5 Comparing Interaction Methods

The run time and memory scalability results for generated state machines with n actions and $n^2 - n$ actions are presented in the plots in Figure 27. As one might expect, it is seen that only having a switch statement in the event handler is the most efficient memorywise for low amounts of actions in the n actions plot. It increases the least in time between states as well which makes it scale the best. Next up is the addition of a mediator which performs slightly slower than the pure switch statement with global variables. Message passing gives the slowest run time and scales the worst. Memory size of the state machines are by far at the highest using 3D arrays for any case. The other interaction implementations are closer with message passing taking up the most space and global switch using the least.



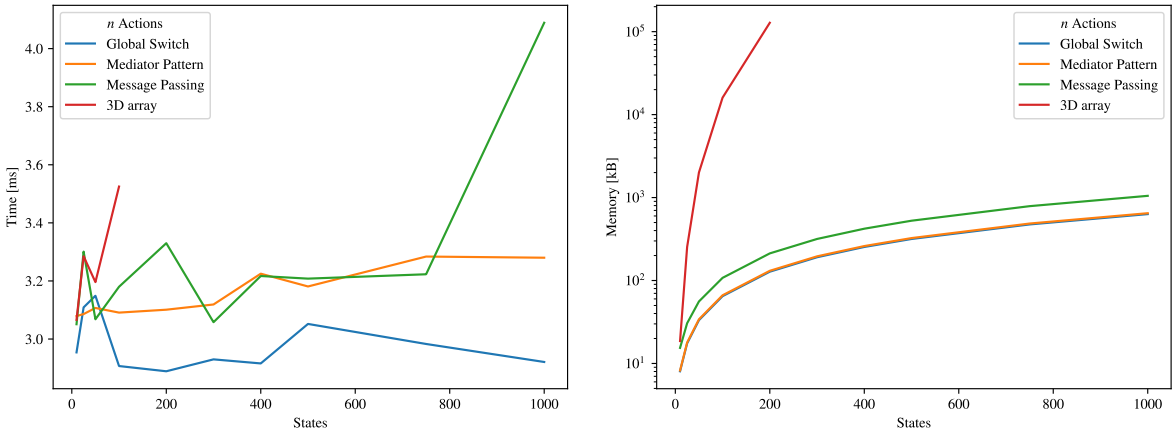
(a) Time n **(b)** Size n



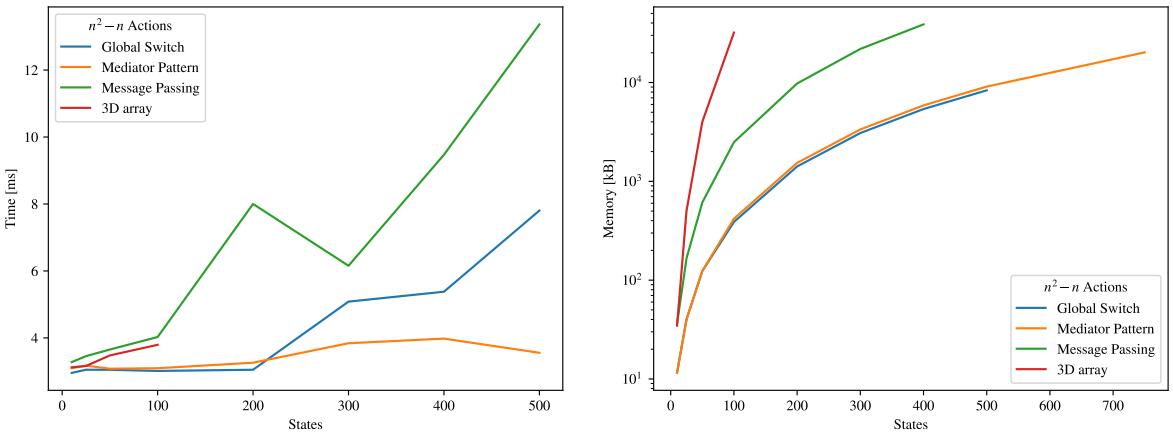
(c) Time $n^2 - n$ **(d)** Size $n^2 - n$

Figure 27: Performance for different interaction methods compiled without optimisation flags.

If we look at the results when including optimisation flags instead in Figure 28, the message passing method is more on par with using a mediator, until reaching 1000 states for n actions. The Global Switch approach still is the fastest and the 3D array implementation does not seem to perform well with the limited data points. In the memory department, the 3D array approach is still the worst by a great margin, with the Message Passing, Mediator and Global Switch approach being closer to each other. For $n^2 - n$ actions the switch performs worse than the mediator pattern, but the measurements are relatively noisy. The memory required in this case using message passing is higher with optimisation, and the mediator and switch approach is still close.



(a) Time n **(b)** Size n



(c) Time $n^2 - n$

(d) Size $n^2 - n$

Figure 28: Performance for different interaction methods compiled with the -O2 optimisation flag.

5 Discussion

5.1 State Machine Implementations

When looking at the results from the scalability benchmarks and focusing primarily on run time and memory size, three of the implementations stands out: Nested Switch, Array of Structs, and Basic State Pattern. The results depend on what type of optimisation the state machine will be subjected to, and whether run time or memory usage is of top priority.

When running without optimisation flags, the Array of Structs, and Basic State Pattern implementations results in similar run times, with a slight edge for Array of Structs when considering larger state machines. The Array of Structs implementation also turns out to be the most memory efficient, hence we can consider the Array of Structs implementation to have the best general performance in an environment without optimisation flags. However, if memory usage is not a priority, the Basic State Pattern implementation might be preferred because of its more robust design pattern.

In an environment with the -O2 optimisation flag, in terms of run time, the Array of Structs implementation lags behind while the Nested Switch implementation catches up to the Basic State Pattern implementation. The Nested Switch implementation just about surpasses the Array of Structs implementation in terms of memory usage, consequently when only considering the best performance the Nested Switch implementation can be considered the winner. However, due to the somewhat inconsistent run time results, and the in our opinion messier code design and more demanding maintenance, the Nested Switch implementation might not be the optimal choice. Instead, the Basic State Pattern might be preferred if run time is the priority, while the Array of Structs implementation might be preferred if memory usage is the priority.

5.1.1 Maintainability

Using the state pattern based design requires more memory than some of the more straightforward, non-pattern based approaches, but it provides some advantages in terms of design. Since the state-specific behavior is encapsulated, the code is more modular and maintainable. The states are all independent from one another, so adding or changing a state is simple. Changing and adding transitions is also easy, boiling down to adding a function pointer in the specific state struct if using the "Basic State Pattern" method or changing an entry in the matrix if using the "State-Table Pattern". The transition logic used lessens the need for long conditional statements used in the Nested Switch approach. In the Array of Structs and Function Pointers implementations the basic transition logic is assembled in a structured way and all additional transition logic is contained within the `eventHandler` functions. In our opinion, this makes the code both readable and maintainable, especially with a tool such as the interactive code editor. For more complex state machine specifications however, states may need to know about each other which makes the code harder to understand and maintain as state become more coupled. There is a slight learning curve to pattern-based design as well. This is amplified when introducing hierarchy, history and concurrency in the Hierarchical State Pattern that need even more structure to correctly model complicated behavior. If memory or code size is of importance, the state pattern can be problematic due to the general structure needed to encapsulate state specific behavior. In the Basic State Pattern and Hierarchical State Pattern, there is an explosion of structs, one for each state, and the extra for the interfaces and other functionality. Compared to the Array of Structs and Function Pointers implementations that use enumerate types for states and events, this is a notable difference in data.

5.2 Case Study

The results from the case study shows that the Array of Structs implementation performs the best in the no optimisation flags environment with regards to both run time and memory usage. When using the -O2 optimisation flag the Basic State Pattern implementation results in the fastest run time while the Nested Switch implementation needs the least amount of memory. These results are inline with the results from the scalability benchmarks, and the same motivation can be used again to discourage using the messier Nested Switch implementation because the performance difference is small for state machines of this size.

5.3 State Machine Interaction

While noise is hard to fully eliminate in any setting, it was highly apparent in these measurements, which hurts the validity of these run time results. A run time measurement for a lower amount of states should be either the same or less than the run time for a higher amount of states, except in some special cases. We tried retiming these outliers, but it did not really give any improved results. In general there are problems with comparing interaction methods in a sequential setting. First of all, in some way, you need to check the state of the other state machine when performing the action. Unless this is handled through a matrix/table, this will always boil down to an if/switch statement somewhere in the code, most often the event handlers. This will make the Global Switch approach the obvious winner always when comparing run time of the interaction methods since it only checks the state with a switch statement without any communication which works only when the other state machine object is a global object. The other implementations barring the 3D array one does the same but with some added means of communication. Therefore, it might be better to look at the results as a sort of performance loss when adding extra communication. For all the implementations, the Basic State Pattern approach from the previous benchmark was used as the base state machine for the interacting state machines. This might be less beneficial for some of the interaction methods.

Judging from the results for n actions the message passing approach is the slowest out of the three, due to the communication overhead in form of the extra messaging functions and message queue array needed to be accessed every time an event happens. The 4th approach using 3D arrays has close to no results making it impossible to comment on. It could be an interesting approach to use for smaller sized state machines but even then the memory usage is so high that one would be better off using one of the other approaches or a hash table. In the case of using the -O2 optimisation flag the interesting case is how the message passing and mediator pattern seem to compete in run time. Results for the $n^2 - n$ case show a similar trend. It suggests that using a mediator performs better then the Global Switch method by a great amount in some points, when the difference should at best be small if not negligible which is seen at the lower states.

5.3.1 Maintainability

The global state machine variables with the switch statement for interaction performs the best both in terms of run time and memory size but there is a tradeoff in terms of software design and maintainability of the code. As always with switch statements, for larger, complex state machines they become longer and harder for future developers to understand. On top of that, having the context objects being global makes the code more difficult to maintain due to there being a tighter coupling between the context object and transition functions since it expects the global variable to be known.

As previously mentioned with the mediator, it helps decouple dependencies in the code and simplifies interaction as all communication goes through the mediator. This makes the code easier to maintain as the components of the code are then more independent, and makes them easier to update and debug. For simpler state machines as the one worked on in this report, this works well but if the state machine is more complex, the mediator can quickly gain too much functionality and responsibilities, turning into what is called a god object.

Message passing for interaction has the benefit of making more modular code, as the event handlers of the state machine only use relevant messages to change state and work well independently if there are no messages. The real problem arrives when managing a great size of messages, if not done well there are scalability and performance issues by the communication overhead, which was seen in the results.

Using an array to handle the interactions provides more structure and is easy to comprehend. For large state machines, maintaining the array can prove to be a challenge with the increasing size of the arrays. If one would need to add an event or state for a complex state machine the structure of the arrays would need to be changed. While more structure can be provided with this approach there is great memory issue as the results suggested.

5.4 Iterative Code Editor

The iterative code editor allows for easy updates of the Array of Structs implementation through console commands. It ensures that the structure of the implementation stays intact and reduces the risk of bugs outside of the `eventHandler` functions. Overall, it is a simple precaution to ensure safe programming.

6 Conclusions

In this report, the main goal was to suggest a way of working with state machine from model-to-code following some different implementations of state machine design and compare these in the form of a benchmark, looking at performance primarily. The second part suggested a way to iteratively update the generated code and performance measures for interaction between two sequential state machines.

The suggested way of working with state machines was to model a given specification in the UML modeling language, use the XML/XMI output file from the model in a script (in our case written in Python) that parses and finds the components, and writes the C code following an implementation. Implementations suggested were taken from related work in the area and traditional implementations. They consisted of the Nested Switch, Function Pointers, Array of Structs, Basic State Pattern, State-Table Pattern and Hierarchical State Pattern implementations. Run time of the implementations were measured by running the generated state machines of different sizes through 10^5 randomly generated valid events and run on simple state machines consisting only of states and transitions. Memory was measured as the size of the source file containing the implementations, and the implementations were compiled both with and without the -O2 optimisation flag. The results favoured the Array of Structs and Basic State Pattern implementations and the choice depends on the optimisation used and the priority between run time and memory.

An iterative code editor was implemented for the Array of Structs implementation, it allows the user to update the basic structure of state machine code without the risk of introducing bugs.

Run time results from adding interaction between two state machines, mainly for state machines implemented with the state pattern design had using the mediator pattern for communication as the best option. Declaring the state machines as global variables and using a switch statement was treated as the default approach since it adds no extra communication overhead to the code, and while it performed better than using a mediator it has big challenges in regards to code design (modularity, maintainability) and is most likely not used in practice. The second best option was therefore the mediator pattern both in run time and memory size. Other proposed ideas consisted of using message passing and a 3D array of function pointers.

7 Future Work

7.1 Code Generator Functionality

The code generators can be expanded with support for additional features. One of such is generation of unit tests and model validation tests. Unit tests makes sure individual units of the state machine works correctly, with an example being verifying transitions working as intended. One way this could be done is generating assert statements in the C code that checks the active state after a transition. Model validation instead looks at the context of the whole system and ensures that it behaves correctly.

7.2 State Machine Structure

The analysis conducted here only looked at the case of "flat" state machines, with only states and transitions caused by external events. A similar benchmark could be done for other types of state machines with added features (e.g. composite states, orthogonal regions) if that is what the systems use.

7.2.1 Event and Transition Logic

In this project we have treated each state transition as a separate external event. In practice there might be different state transitions triggered by the same external event depending on the current state. It could be interesting to look at how reducing the number of events for the same number of transition would effect the performance of the implementations. It would reduce the number of events but probably require some sort of conditional statement directing the external event to the correct transition.

7.3 Maintainability

The iterative code editor can help with maintainability by adding and deleting states and events in the code part of the state machine. A truly maintainable state machine could go a step further and enable code-to-model transformation by update the graphical state machine model with regard to changes in the code.

7.4 Parallelization

Communication between state machines were conducted here with the state machines run sequentially. It could be interesting to look at parallelization for systems with several state machines and see how well it performs in terms of speedup and scaled speedup.

7.5 Additional Implementations

The analysis conducted here used six different implementations for state machine design, and four different ways to handle interaction between two state machines. Naturally, there are more implementations out there to try. The state pattern is only one of several different design patterns in software design [10]. If state machines with more functionality are of interest, most modern research covers proposals on how to implement the full semantics of UML specifications. Another idea is to test the performance of the generated code from this analysis with the generated code from available tools.

References

[1] Microsoft 365. Visio. https://www.microsoft.com/sv-se/microsoft-365/visio/flowchart-software/. (accessed: 2023-05-22).

[2] A. Kumar. How to implement finite state machine in C. *aticleworld.com*, August 2017.

[3] A. Myers. State machines. *CS 211*, May 2006.

[4] David Harel. Statecharts in the making: a personal account, June 2007.

[5] E. Doherty. What is object-oriented programming? OOP explained in depth. *educative*, April 2020.

[6] E. Domínguez, B. Pérez, Á. L. Rubio, and M. Zapata. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54:1045–1066, Oct 2012.

[7] Eclipse Foundation. Eclipse Papyrus ™. https://www.eclipse.org/papyrus/. (accessed: 2023-05-22).

[8] Python Software Foundation. xml.etree.ElementTree — The ElementTree XML API. https://docs.python.org/3/library/xml.etree.elementtree.html, 2023 (accessed Feb. 2023).

[9] I. Jacobson G. Booch, J. Rumbaugh. *Unified Modeling Language User Guide, The*. Addison Wesley, 1998.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., December 1994.

[11] A. S. Gillis. What is model-driven development (MDD). *TechTarget*, June 2018.

[12] S. Kumar. Mediator design pattern. *GeeksforGeeks*, September 2022.

[13] I. A. Niaz and J. Tanaka. CODE GENERATION FROM UML STATECHARTS. *Proceedings of the IASTED International Conference on Software Engineering and Applications*, 7, Oct 2003.

[14] OMG. XML Metadata Interchange (XMI) Specification - Version 2.5.1. *OMG.org*, June 2015.

[15] OMG.org. ABOUT THE UNIFIED MODELING LANGUAGE SPECIFICATION VERSION 2.5.1. https://www.omg.org/spec/UML/2.5.1/About-UML. (accessed: 2023-05-04).

[16] OMG.org. MISSION & VISION. https://www.omg.org/spec/UML/2.5.1/About-UML. (accessed: 2023-05-04).

[17] OMG.org. OMG Unified Modeling Language™ (OMG UML) Version 2.5.1. *UML.org*, December 2017.

[18] Visual Paradigm. About US. https://www.visual-paradigm.com/aboutus/. (accessed: 2023-05-22).

[19] S. Lignos. Working with State Machines in Angular. *medium.com*, June 2019.

[20] A. V. Saúde, R. Victório, and G. Countinho. Persistent State Pattern. *PLOP '10: Proceedings of the 17th Conference on Pattern Languages of Programs*, pages 1–16, October 2010.

[21] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20:19–25, September 2003.

[22] V. Spinke. An object-oriented implementation of concurrent and hierarchical state machines. *Information and Software Technology*, 55:1726–1740, March 2013.

[23] E. V. Sunitha and P. Samuel. Automatic Code Generation From UML State Chart Diagrams. *IEEE Access*, 7:8591 – 8608, jan 2019.

[24] UML.org. INTRODUCTION TO OMG'S UNIFIED MODELING LANGUAGE™ (UML®). *UML.org*, Jul 2005.