Master Thesis

Behavioral modelling of embedded software using execution traces

Satej Vinayak Khandeparkar

School of Information Technology

Halmstad University, July 5, 2017
The woods are lovely, dark and deep.
But I have promises to keep.
And miles to go before I sleep.
And miles to go before I sleep.

— Robert Frost
ABSTRACT

Software updates made by developers often achieve their intended purpose, but these updates may also lead to an anomalous behavior previously unknown to the developers. This might be due to their interaction with other parts of the system. If the developers had a tool which could help them to visually see these changes as a behavioral model, it would benefit them to actually know how the changes have affected the behavior of the system. Thus, empowering them to fix any side effects or bugs that arise as a part of their update.

Thus, in order to visualize and compare learned behavioral models, a tool was created which would model the behavior from traces generated by scenarios based on the related work in the area of inferring models of software systems. This tool was specifically intended for embedded software. So, to compare changes based on updates and functional changes of embedded software, behavioral models of scenarios were obtained for different versions of a Real Time Operating System (RTOS) Kernel. The visual comparison algorithm proved to be effective in visualizing the differences between behavioral models for a particular scenario across the versions.
ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Nicholas Wickström, for his meticulous support, exhaustive suggestions and thorough guidance. I would like to thank my examiner, Dr. Mohammad Mousavi, for his invaluable advise. I would also like to thank doctoral student Mahsa Varshosaz for her exclusive contribution. A big thank you to all those involved directly and indirectly. Last but not least to Halmstad University, CERES and CAISR research environment which made it possible for this work to reach fruition.
# CONTENTS

1 **INTRODUCTION** 1  
1.1 Behavioural model 2  
1.2 Motivation 3  
1.3 Research Question 4  
1.3.1 Research Goals 4  
1.4 Organization of the chapters 4  

2 **PRELIMINARIES** 7  
2.1 Definitions and their usage 7  
2.2 Types of state machines 8  
2.2.1 FSM 8  
2.2.2 EFSM 9  
2.3 Timed Automata 9  

3 **RELATED WORK** 11  
3.1 Learning Automata 11  
3.2 Inference of FSM 11  
3.3 Inference of EFSM 12  
3.4 State merging Algorithms 13  
3.5 Positioning of the work 13  

4 **METHODOLOGY** 17  
4.1 Final model type 17  
4.1.1 Choices for the final model type 17  
4.2 Techniques 18  
4.2.1 Choices related to techniques 18  
4.2.2 Instrumentation of the code 19  
4.2.3 Instrumentation of the RTOS kernel functions 20  
4.2.4 Wrapper functions for instrumentation of RTOS kernel methods 21  
4.3 Trace generation 23  
4.3.1 Choices related to trace generation 23  
4.4 Scenarios used in this work 25  
4.4.1 RTOS functions used for the Scenarios 25  
4.4.2 Communication Scenarios 26  
4.4.3 Task Scenarios 26  
4.4.4 Timing Scenarios 26  
4.5 Analysis of the traces 27  
4.5.1 Choices related to analysis of traces 27  
4.5.2 Abstraction 28  
4.6 Merging Algorithm 30  
4.6.1 GK- tails 31  
4.6.2 EFSM 31  
4.6.3 MSM Algorithm 34  
4.6.4 Choices related to merging algorithm 35
CONTENTS

4.7 Custom Merge Algorithm 37
4.8 Visual Comparison 40
4.9 Development stages of the tool 42
4.10 Summary of choices 43

5 RESULTS 45
5.1 SMT 1.0 45
5.1.1 Step 1: Generate a random sequence 47
5.1.2 Step 2: Instrumentation of functions 47
5.1.3 Step 3: Abstract the states 47
5.1.4 Step 4: Initial behavioral model 47
5.1.5 Step 4: Final behavioral model 48
5.2 SMT 2.0 50
5.2.1 Step 1: Instrument the functions 51
5.2.2 Step 2: Read the trace 51
5.2.3 Step 3: Abstract the states 51
5.2.4 Step 4: Creating an initial graph 51
5.2.5 Step 5: Merging the initial graph 52
5.2.6 Step 6: Filtering the merged state machine 52
5.2.7 Step 7: Displaying the final behavioral model 53
5.2.8 Comparison of behavioral models 54
5.3 Adapting SMT 2.0 to the RTOS kernel 55
5.3.1 Generation of traces 55
5.3.2 Creating a initial graph 56
5.3.3 Merging the initial graph 56
5.3.4 Displaying the final behavioral model 57
5.4 Result for sub-goal 1 58
5.5 Interpretation of results in the following sections 59
5.6 Results for sub-goal 2 60
5.6.1 Results from RTOS kernel with 4 versions 60
5.6.2 Results from RTOS kernel with 9 versions 62
5.7 Results for sub goal 3 64
5.7.1 Result from Kernel with 4 versions 64
5.7.2 Result from Kernel with 9 versions 67

6 DISCUSSION 69
6.1 Verification of behavioral model from a known model 69
6.2 Modelling Non-functional characteristics 69
6.3 Precise identification of methods that have functionally changed 70
6.4 Randomization 70
6.5 Enhancing SMT 2.0 using Machine Learning 71
6.5.1 Approach to utilize the clustering technique for SMT 2.0 71

7 CONCLUSIONS AND FUTURE WORK 73
7.1 Future Work 73
ACRONYMS

VCS  Version Control System
API  Application Programming Interface
FSM  Finite State Machine
EFSM  Extended Finite State Machine
RTOS  Real Time Operating System
DFA  Deterministic Finite Automaton
NFA  Non-Deterministic Finite Automaton
PTA  Prefix Tree Acceptor
INTRODUCTION

A model is defined as a description of observed or predicted behaviour of a system, simplified by abstracting certain details [1]. Models allow complex systems, both existent and merely specified, to be understood and to predict their behaviour [1]. Thus, a model that describes the observed or predicted behaviour of a software is called a software model.

Software models are categorized in two types, those that describe structure, i.e., they tell what a software is, e.g., class diagrams as shown in Figure 1, and those that describe behaviour, i.e., they tell what a software does, e.g., state machines as shown in Figure 1 [2].

![Example of a class diagram](image1)

![Example of a state machine](image2)

Figure 1: These figures illustrate the example of class diagram and a state machine.

Software development is a process where specifications are converted to structure and software is programmed according to that structure. This work is related to doing exactly the reverse. As it is illustrated in Figure 2, the model is generated from a program that is already developed using a set of traces that are acquired from a set of scenarios considered for the program. The generated model describes the behaviour of the software for that particular scenario.
1.1 BEHAVIOURAL MODEL

As the model describes behaviour of the system, in this section what constitutes as behavior is explained.

A queue is an abstract data structure in which elements are kept in order. Addition of elements is done by a method called `enqueue()`, and deletion of elements is done by `dequeue()`. When a upper limit is imposed on the number of elements a queue can store, it is called as a bounded queue [3].

The behavior of a bounded queue is such that when data is enqueued, the queue size increases and when data is de-queued, the queue size decreases. The state of the queue is characterized depending upon the size of the queue, it signifies whether the queue is empty or not empty, or if the queue is full.

Consider a bounded queue data structure of max size 3. Figure 3 shows the model describing behaviour of the bounded queue for a scenario where `enqueue()` was executed followed by `dequeue()`, and continued by `enqueue()` even after the queue was full.
From Figure 3, it is observed that when the queue is created it is in **Empty** state, since it does not contain any data.

When a call to `enqueue()` is made, a single data element is inserted inside the queue, increasing the queue size to 1. This makes the queue change state from **Empty** to **NotEmpty** state as shown in Figure 3.

A call to `dequeue()` removes a data element from the queue which bring the state from **NotEmpty** back to **Empty** state. 
Again a call to `enqueue()` makes the queue size equal to 1 and thus the queue enters the **NotEmpty** state. The next call to `enqueue()` inserts another data element in the queue, but since, queue size is not equal to max, the state is still **NotEmpty**.

The following call to `enqueue()` makes the queue size equal to 3 which is the max size and thus the queue goes from **NotEmpty** state to **Full** state. Once the queue has reached its max size, further data elements cannot be inserted and if tried as shown in Figure 3, an illegal state is reached.

### 1.2 Motivation

Every software has shortcomings and flaws. To overcome those shortcomings and flaws, new versions are released. The reasons for new versions varies, for eg. adding a new feature, patching a minor bug or patching a major security issue[4].

Bugs in a software arise when a developer’s intent is not correctly implemented as code. If the developers are provided with a model of their code it could be helpful for recognition of anomalies and correcting them [5]. Changes in the source code can also be seen by taking a diff of two versions. But doing so may highlight only syntactical changes [6]. If impact of functional changes made to the code could be visually seen, it may assist in spotting anomalies.

Kothari et al. have not explored their claim that models can help developers spot anomalies, since their main focus was on learning the software model using FSMGen, rather than comparison of models. Part of the claim about comparison is further explored in this work by comparing behavioral models of scenarios of different versions of a RTOS Kernel. The RTOS kernel specifications are described in Chapter 9. Detailed review of the related work and a comparison of the relevant tools is mentioned in Chapter 3.

Problems such as an update or a functional change leading to side-effects or causing a malfunction can even be present at a small scale. Also embedded software is inherently state-full, where, the current
output depends on the input and its internal state [7]. Thus, a Real Time Operating System Kernel was chosen for this purpose, which had been developed through number of iterations, before reaching its final version.

1.3 RESEARCH QUESTION

The research question considered in this work is that, can behavior changes of an embedded software with different versions be traced by comparing the behavioural models obtained from execution traces?

1.3.1 Research Goals

The following sub goals are directed towards the higher ideal of comparing behavioural models.

Based on the overarching goal of comparing behavioral models to detect changes, a set of sub goals has been identified:

1. Develop a method to visually compare the obtained models to highlight changes in the form of different transitions.

2. Analyze the behavioral changes by visually comparing the behavioral models and difference of the identified functionalities from source code.

3. Investigate mechanisms for analyzing indirect behavioral changes, e.g. consider a function which calls a method X to accomplish certain functionality. The influence of method X on the behaviour of embedded software, refers to analyzing indirect behavioral change.

1.4 ORGANIZATION OF THE CHAPTERS

The remaining master thesis is organized as follows.

- Chapter 2: Preliminaries.
  
  This chapter contains the most important terms and their definitions that are used throughout the thesis.

- Chapter 3: Related Work.
  
  This chapter describes in detail the reviews of the work and literature related to this thesis. It also compares the relevant tools that have been mentioned in the related work.
• Chapter 4: Methodology.
  This chapter describes the features included in SMT 1.0 and 2.0. It mainly focuses on the choices and the decisions that were taken when building the tools in the process of achieving the research goals.

• Chapter 5: Results.
  This chapter deals with the results obtained in every phase of the thesis.

• Chapter 6: Discussion.
  In this chapter further improvements for the tools and different ideas have been discussed.

• Chapter 7: Conclusions and Future Work.
  This chapter is regarding the conclusions formulated from the results obtained in harmony with the research goals. It also mentions the direction that could be taken for the work to progress further.

• Bibliography
  Contains all the references studied and websites accessed.

• Appendix A
  This appendix contains details related to the bounded queue API, which was modelled using SMT 1.0.

• Appendix B
  This appendix contains details related to the RTOS kernel functions.

• Appendix C
  This appendix focuses on the architecture and libraries utilized for building SMT 2.0.
PRELIMINARIES

This chapter contains definitions of terms and terminology utilized in this thesis.

2.1 DEFINITIONS AND THEIR USAGE

State-full and state-less system definitions are mentioned to highlight the importance of embedded software as it is inherently state-full system [7].

The terms such as inspectors, mutators and concepts such as instrumentation, abstraction and traces are used when tool development is considered.

The definitions of FSM, EFSM and Timed Automata are used to decide the type of final model that is used to represent the behavior of a system.

Definition 2.1.1. State-less system

A system is considered state-less when its current output depends on the current input. Since UDP, a transport layer protocol from the TCP/IP stack does not maintain a connection between client and server, it is considered as a state-less system [7].

Definition 2.1.2. State-full system

A system is considered state-full when its current output depends on the current input and internal state. TCP, a transport layer protocol from the TCP/IP stack maintains a connection between client and server, thus it is considered as a state-full system. RTOS is also an example of state-full system [7].

Definition 2.1.3. Inspectors Any function which does not lead to a change of state after execution is classified as an inspector function [8].

Definition 2.1.4. Mutators Any function which changes the state after execution is classified as a mutator function [8].
Definition 2.1.5. Instrumentation Wrapping of executing inspector functions before and after the mutator function is referred to as instrumentation in this context [8]. A simple example of instrumentation is monitoring the size of a queue data structure. Execute \texttt{size()} method and store the output in a list. Then execute the \texttt{enqueue()} method to add an item to the queue. Execute \texttt{size()} method again and store the output in a separate list. This way the \texttt{enqueue()} method is said to be instrumented.

Definition 2.1.6. Traces An event is a tuple \((l, v)\), where \(l\) is the name of a function and \(v\) is a mapping from parameter variables for \(l\) to concrete values. A trace \(t \in T\) is a finite sequence of events, often written as \(\langle (l_0, v_0), \ldots (l_n, v_n) \rangle\) [9].

Definition 2.1.7. Abstraction The process of analyzing raw information contained in the traces, obtained as a result of instrumentation, by mapping the raw information to abstract states based on observed criteria, is referred to as abstraction. For e.g., \(\text{size} = 0\) is mapped to \text{Empty} state, \(\text{size} = \text{max}\) is mapped to \text{Full} state [8].

2.2 TYPES OF STATE MACHINES

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

2.2.1 FSM

Definition 2.2.1. A Finite State Machine (FSM) can be defined as a tuple \((S, s_0, F, L, T)\). \(S\) is a set of states, \(s_0 \in S\) is the initial state, and \(F \subseteq S\) is the set of final states. \(L\) is as defined as the set of labels. \(T\) is the set of transitions, where each transition takes the form \((a, l, b)\) where \(a, b \in S\) and \(l \in L\) [9].

Figure 4: This figure illustrates a Finite State Machine
Figure 4, is an example of a Finite State Machine. There are two states: State A and State B. The edges are annotated with the functions, Command_on() and Command_off().

2.2.2 EFSM

**Definition 2.2.2.** An Extended Finite State Machine (EFSM) is a tuple $(S, s_0, F, L, M, \triangle, T)$, where $S$, $s_0$, $F$ and $L$ are defined as in a conventional FSM. $M$ represents the memory of the EFSM – a collection of variables. An assignment of variables in the memory to concrete values is denoted by $m \in M$, where $m$ is the concrete value and $M$ is the variable. The update function $\triangle$, is represented as $L \times M \to M$. The set of transitions $T$ is an extension of the conventional FSM version, where transitions take the form $(a, l, m, b)$, where $a, b \in S$, $l \in L$, and $m \in M$ [9].

![Figure 5](image.png)

Figure 5: This figure illustrates a Extended Finite State Machine

Figure 5, shows an example of an Extended Finite State Machine. The nodes are represented as states. The edges are annotated with functions and the value of the variables. When Command() is executed with variable $c = 1$, the system transitions from state A to state B. The added information regarding data associated with the function’s input variables, makes it Extended Finite State Machine [10].

2.3 Timed Automata

Timed automata is a finite automata with transitions annotated with timing information. The timing information is in the form of guards, which control the behaviour of the automaton. This type of model is best suited for analyzing timing behaviour of software for safety checks in application domains such as aeronautics, aerospace, automotive [11].

Formally, a timed automaton is a tuple $A = (Q, \Sigma, C, E, q_0)$, where $Q$ is a finite set. The elements of $Q$ are called the states of $A$. $\Sigma$ is a finite set of alphabet or actions. $C$ is a finite set representing clock. $E$
$\subseteq Q \times \Sigma \times B(C) \times P(C) \times Q$ is a transition relation, where $B(C)$ is the set of boolean clock constraints involving clocks from $C$, and $P(C)$ is the powerset of $C$. $q_0$ is an element of $Q$, called the initial state. An edge $(q,a,g,r,q')$ from $E$ is a transition from state $q$ to $q'$ with action $a$, guard $g$ and clock resets $r$ \cite{11}.

![Timed Automata Diagram]

Figure 6: This figure illustrates a timed automata

In Figure 6, for transition $t_2$ to take place, it has to be done after 3 time units, $x > 3$ is the guard that controls this condition. For transition $t_3$ to take place, it has to be done within 7 time units, $x \leq 7$ is the guard that controls this condition. If transition $t_1$ is taken, then the system enters $S_1$ state with clock being reset by $x := 0$ \cite{12}. 


RELATED WORK

This chapter is about the literature survey done related to learning behavioral models of software.

3.1 LEARNING AUTOMATA

The following paper was considered as it was one of the earliest work for learning automata.

Angluin in her paper [13] presents L* algorithm which learns an unknown regular set over an alphabet from a *minimally adequate teacher* efficiently. The *minimally adequate teacher* is a source of examples and learning algorithm is the learner. The *minimally adequate teacher* has to answer questions related to membership queries from the learner when the learner asks, if a string t is a part of a unknown regular set. Also, when the learner comes up with a description of a regular set, the *minimally adequate teacher* has to say yes, if the regular set matches the unknown set, and if the answer is no, then the *minimally adequate teacher* generates a counter-example which starts the process again.

The concepts of *minimally adequate teacher* and generation of a counter-example are experimented with, in this work.

3.2 INFERENCE OF FSM

The following papers were referred since they used two different approaches to arrive at their end result which was an FSM.

Dallmeier et al. created a tool named ADABU which mined finite state machines as behavioural models from executions of functions. They classify the functions as *inspectors* which do not change the state of the system, and *mutators* which change the state of the system after execution. They capture the state of the object by using inspector functions which they select after analysing the source code using static analysis.

En-framing refers to calling the inspectors before and after the mutator functions. After finding the inspectors using static analysis they instrument the code of mutator functions by en-framing them with the inspector functions. They use this novel method to capture the data which helps to characterize the states as Empty or NotEmpty instead of using random values for the states [8].
The concept of characterizing the states from captured data through instrumentation is invaluable to this work.

Kothari et al. have created a tool named FSMGen which learns an FSM using symbolic execution technique and predicate analysis. They say that error arises when there is a friction between programmer’s intent and resulting code. They claim that if behavioural models of tinyOS programs are given to the programmers, they can find changes in behaviour and analyse the changes. They analyse only one version of tinyOS programs and thus do not support how does a change look like while a software is under development [5].

The work done in this master thesis explores their claim partly, where there arises a need to compare two models, by generating behavioural models for different versions of the same software from execution traces and visually comparing them.

The visual comparison algorithm mentioned in Chapter 4 of this work, helps in spotting the changes between versions of the software, and thus enables to visually see the changes, when the software was under development.

3.3 Inference of EFSM

The following papers described their tool flow and how to derive an EFSM from recorded traces.

Lorenzoli et al. in their paper mention that FSMs fail to capture the relationship between data and component interaction. Where, data influences which function to execute next or which path to consider in a conditional statement. Thus important information and behaviours which are data dependent are lost [10].

They present GK-tails an inference algorithm which combines the k-tails with Daikon. Daikon operates on a set of \((\text{variable}, \text{value})\) pairs and generates an invariant such as \(v_1 \leq x \leq v_2\). Where \(x\) is the variable, \(v_1\) and \(v_2\) are the upper and lower bounds respectively. Daikon assists in finding data invariants and thus annotates the FSM with data guards on the transitions. [14, 15, 10, 16].

They explain the GK-tails algorithm and compare 3 different merging criteria which they claim to be suitable for different quality of test suits. They also mention that since the EFSM generated is dependent on traces obtained from the test suits, the quality of test suits are responsible for quality of models learned [10].

Walkinshaw et al. present the EFSM inference algorithm that incorporates machine learning techniques which assist in enhancing merging quality and producing a deterministic model. Each software generates traces which contain data unique to that particular software.
They present that different machine learning algorithms can be applied depending on the uniqueness of the data generated by the software to be modelled [16].

Both the papers mentioned above were crucial as it gave an insight into the concepts of state merging and using machine learning to enhance the merging process. Also, the addition of data to the final behavioral model, was a concept worth exploring.

3.4 STATE MERGING ALGORITHMS

These papers were considered to particularly study state merging algorithms.

Lambeau et al. present MSM algorithm which extends the basic merging algorithm using the concept of state coloring and state labeling. State coloring helps to avoid merging incompatible states where as state labeling helps in merging compatible states. MSM algorithm uses domain knowledge to further enhance the merging process [17].

Dupont et al. propose a query-driven state merging algorithm (QSM). In their work RPNI and Blue-Fringe algorithms are extended by submitting to the end-user membership queries. They also consider domain knowledge in the form of goals or domain properties which further constraint state merging tasks and also reduces the number of queries [18].

The idea of state labeling constraints from MSM algorithm was beneficial as it could be used for the merging process when coupled up with characterization of states.

3.5 POSITIONING OF THE WORK

In this work, four tools were studied for their features and techniques. Source code of EFSM tool was available as it is an open source project [16], rest of the tools, such as FSMgen [5], Adabu [8], tool mentioned in [10], were only described in their respective papers.

The tool mentioned in [10] is referred to as AGSBM, which is an acronym of the paper’s title, since the authors did not mention any name for their tool.

From Table 1, all four tools are compared based on a category and the methods within that category. The categories were decided based on the flow/working of the tools as described in their respective papers.

Comparison starts with the techniques used by the tools, where only FSMgen uses symbolic execution and predicate analysis, whereas all other tools use traces.
14 RELATED WORK

<table>
<thead>
<tr>
<th>Category</th>
<th>Method</th>
<th>FSMGen</th>
<th>Adabu</th>
<th>AGSBM</th>
<th>EFSM tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Traces</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technique</td>
<td>Symbolic execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace generation</td>
<td>Scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace generation</td>
<td>Randomization</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traces</td>
<td>Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traces</td>
<td>Machine Learning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>Modified K-tails</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>EFSM with ML</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>MSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>Custom</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Model type</td>
<td>FSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Model type</td>
<td>EFSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparison</td>
<td>Visual Comparison</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: This table represents which methods were employed by the tools mentioned in the related work section.

The method used for trace generation is scenario based for FSMgen, Adabu and AGSBM. Random execution of the functions or randomization for generation of random scenarios was not utilized by any tool, hence the feature is highlighted in yellow.

Traces undergo abstraction in Adabu for classification of states, whereas EFSM tool uses machine learning classifiers.

When it comes to merging, AGSBM uses the GK-tails inference algorithm [10]. EFSM tool modifies the GK-tails inference algorithm to utilize the machine learning classifiers for the merging purpose. They refer to it as EFSM inference algorithm [16]. Since, the MSM algorithm [17] was not utilized by any tool it is highlighted in yellow.

The final inferred model is displayed as an FSM by FSMgen and Adabu [5, 8]. EFSM tool and AGSBM display the final inferred model as an EFSM [16, 10].

In the comparison category, the option to visually compare two different behavioral models was not present in the tools studied as a part of the related work in the area of learning behavioural models, and thus this method is highlighted in yellow. Except for EFSM tool all other tools require source code for learning the model.

From Table 1, it is evident that there are gaps as far as utilizing randomization of functions for creating random scenarios and visually comparing the behavioral models are concerned.
Based on the related work in the area of learning models of software systems, Software Modelling Tool (SMT) 1.0 and 2.0 were created. This tool models the behavior of the scenario from the traces obtained after executing the methods in the scenario.

SMT 1.0 experiments with randomization and SMT 2.0 explores the comparison of behavioral models at a small scale pertaining to a RTOS kernel.

Although the related work was about learning the behavior model, the tools were created to model the behavior, since this work was started from scratch and intended to go in the direction of comparison of learned models.
**METHODOLOGY**

This chapter is about the methods that were chosen to design the tool to generate a behavioral model of a RTOS kernel. The following sections describe the choices and the features that were considered in detail. Every section has a short text, which explains the category, followed by choices for that category and their features, followed by motivation for the choice of method and lastly explanation of the method.

At the end of this chapter, the choices of methods for every category are summarized. A link between the development stages and choices is presented.

### 4.1 Final Model Type

As the name suggests, final model type is the way behaviour of a system needs to be represented. How the final model will look like is decided by the type of state machine chosen.

#### 4.1.1 Choices for the final model type

<table>
<thead>
<tr>
<th>Feature</th>
<th>FSM</th>
<th>Timed Automata</th>
<th>EFSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic concept</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Information conveyed</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Effectiveness for depicting behavioral model</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2: This table depicts the choice of final model for representing behavior.

As represented in Table 2, the following features were noted among the choices of FSM, EFSM and Timed Automata.

- **Basic Concept**: This feature is about what an FSM, EFSM and Timed automata represent.
  - FSM represents a state machine with transition annotated along the edges.
  - EFSM represents a state machine with transition as well as data associated with it annotated along the edges.
  - Timed Automata is similar to EFSM but contains only timing data as guards.
• Information conveyed: This feature is about what information is conveyed by the model.
  – FSM conveys component interactions [10].
  – Timed Automata conveys timing information.
  – EFSM conveys relationship between next method to be executed based on data values [16, 10].

• Effectiveness for depicting behavioral models: This feature compares how effective the information in the model is, for capturing behavior of a software.
  – Since data is not involved FSM may not be a better choice for a final model as it may contain inaccuracies [16].
  – Timed Automata is a viable choice since it contains timing information.
  – EFSM contains information related to data associated with the methods and thus it is better suited for a behavioral model.

FSM was chosen in the first stage of development as it would represent states and transitions, data would not be a part of the final behavioral model. Also, the first two features were enough to proceed with this type as the inaccuracies could be tolerated, since, the intended purpose of the behavioral model was not to classify correct/incorrect behaviour [16] but to start with the prototyping of the tool.

EFSM was chosen in the second stage of development as data was possible to be stored along with respective transitions with additional data-structures, and thus could be represented by the final behavioral model.

4.2 Techniques

Technique refers to the primary start point, which will impact the design of the tool. This decision is about the input to the tool. The decision is between, whether to give traces as input, or to give source code as input to the tool.

4.2.1 Choices related to techniques

As shown Table 3, the following features were noted among the choices of Traces and Symbolic Execution technique.

• Ease of implementation: This feature is about the ease with which the technique can be implemented.
## 4.2 Techniques

<table>
<thead>
<tr>
<th>Feature</th>
<th>Symbolic execution</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of implementation</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Instrumentation</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Source code</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Tools</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: This table depicts the choice of technique used for the tool.

- Compared to Symbolic execution technique, generating traces was simpler to implement.

- **Instrumentation**: This feature allows to capture information before and after a method is executed.
  - There was a need to instrument the code for trace generation, there was no such need for Symbolic Execution.

- **Source code**: This feature is about the necessity of source code.
  - EFSM inference tool could learn behavioural model even if only traces were provided as input [16], whereas Symbolic Execution depended on the source code of the software to be modelled.

- **Tool**: This feature is about requirement of external tool.
  - Instrumentation could be performed automatically with a tool as well as manually, whereas Symbolic Execution strictly required a tool which would parse the source code.

Thus from the studied features, the choice was made to go ahead with Traces, as it was simpler to implement although instrumentation had to be performed manually. This choice could lead to a reverse engineering tool in case the source code was not available, and no additional tool was required. Symbolic Execution is a powerful technique and its effectiveness is discussed in Chapter 6.

### 4.2.2 Instrumentation of the code

```
pre_state = [isEmpty(), size(), isFull() ] -> Inspectors
enqueue(5)                    -> Mutators
post_state = [isEmpty(), size(), isFull()] -> Inspectors
```

Figure 7: This figure depicts the procedure of instrumentation.
In Figure 7, the methods isEmpty(), size() and isFull() are referred to as inspectors. The method enqueue() is referred to as mutator. Methods which do not have return type void, do not take parameters as input and have no side effects are chosen as inspectors.

Instrumentation is performed to collect information regarding certain inspector methods before and after a mutator is executed. In Figure 7, it can be seen that before the mutator method enqueue() is executed, the methods isEmpty(), size() and isFull() execute and store their respective outputs in a pre_state list. These same methods are again executed after the call to mutator method enqueue() is performed, their respective outputs are stored in a post_state list.

Thus, whenever a mutator method or the method that is called in a scenario or randomly executed, information regarding that method is stored in pre_state list and post_state list. So, when pre_state list and post_state list, for all the mutator methods are appended together they form a complex_state_list.

This complex_state_list is an input to the abstraction process as described in Figure 11.

4.2.3 Instrumentation of the RTOS kernel functions

Embedded software or generally any software with threading capability makes use of concurrency. Thus, there are context switches that happen between threads, there are also interrupt driven subroutines which make it difficult to capture traces related to the function being executed.

```
trace
0,0,0,0,initKernel,1,0,0,0
0,0,0,0,createMailbox_m_5,1025d8,0,1,0
9,0,1,0,sendWait_s_7,11,0,0,1,recvWait,1,0,1,0 ///-- sendwait response
1,0,1,0 ///-- recvwait response
```

(a) Trace before wrapping loadcontext

```
trace
0,0,0,0,initKernel,1,0,0,1
0,0,0,0,createMailbox_m_1,102bd8,0,1,0
9,0,1,0,sendWait_s_7,1,0,0,1
11,0,0,1,recvWait,1,0,1,0
```

(b) Trace after wrapping loadcontext

Figure 8: This figure depicts the need for instrumenting context switching mechanism specific to embedded software
The trace obtained without instrumenting Loadcontext() results in mixing traces due to a context switch. The trace marked sendwait response in Figure 8, is an example of what mixing of traces means. send_wait() is supposed to block the task from which it is called until the data that is sent to message box, is received by another task.

The trace marked "sendwait response" in Figure 8 contains 9,0,1,0 which is the information before the send_wait() method is executed. It is followed by the name of the method and data associated to the input, "sendWait_s_7" and Lastly information related to the rec_wait() method, starting with 11,0,0,1.

This information related to rec_wait() method is not supposed to be mixed with the trace of send_wait(). This is the problem specific to embedded software as it is a state-full system, where the current state depends on the previous state of the machine. RTOS is a state-full system [7] consisting of context switching and interrupts. They cause mixing of traces as observed in Figure 8.

Thus, this problem of mixing of traces had to be solved. send_wait() and rec_wait() internally called Loadcontext() which would perform the context switch. After Loadcontext() method was instrumented, data corresponding to the send_wait() and rec_wait() method was logged before performing the context switch, this fixed the problem of trace mixing due to context switches. In Figure 8, it can be seen that information of send_wait() and rec_wait() methods is on two different lines. Also, the information gathered before and after the call to respective methods is clearly logged on a per line basis as is seen in Figure 8.

Thus, instrumentation had to be adapted to meet the needs of embedded software which was in this work a Real Time Operating System Kernel.

4.2.4 Wrapper functions for instrumentation of RTOS kernel methods

The code is manually instrumented to capture the traces. Thus wrapper functions are written which are called during execution and the required traces are captured with ease. Instrumentation of the functions using wrappers is performed responsibly to keep the original functionality of the method intact and not introduce any errors.
exception initKernel()
{
    int initK = 0;  // stores return value of init_kernel()
    int isF = NULL; // stores return value of isFull()
    int isE = NULL; // stores return value of isEmpty()
    count = 0;     // stores the length of readyList

    fprintf(ft,"%d,%d,%d,%d, ",initK,isF,isE,count);
    initK = init_kernel();
    count = list_count(readyList);
    fprintf(ft," initKernel,%d,%d,%d,%d
",initK,isF,isE,count);

    return initK;
}

Listing 1: initKernel() wrapper function

Listing 1 shows the code for wrapper function initKernel(). The variables initK stores the return value of init_kernel() method. Variables isF and isE are not useful for this trace and thus are defined as NULL, they are optionally kept, and can be removed as well. The variable count is important as it holds the value for the size of readyList.

It can be observed from the code in Listing 1 that information needed to characterize the state is recorded before and after the call to the function init_kernel() using fprintf() method. The first call to fprintf() is before the call to init_kernel() method. The second call is after the init_kernel() method, the information held in initK and count is logged in a file whose file pointer is ft.

In a similar way, all other functions pertaining to the RTOS Kernel API mentioned in Chapter 9 were wrapped for instrumentation. Also, four additional functions were written to perform effective logging of data for instrumentation, such as list_count() function, since they were not a part of the original kernel API.
4.3 Trace generation

Trace generation is possible due to instrumentation. But to generate traces, methods that have been instrumented need to be executed. After the methods are executed, traces are logged in a text file, which is used as an input to the tool.

4.3.1 Choices related to trace generation

<table>
<thead>
<tr>
<th>Feature</th>
<th>Randomization</th>
<th>Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Purpose</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4: This table depicts the features of randomization and scenarios for trace generation.

As shown in Table 4, the following features were noted among the choices of Randomization and Scenarios.

- Type: This feature is about whether the method to be chosen is automatic or manual in nature.
  - Randomization has the feature of automatic execution of methods, whereas Scenarios is of manual type.
- Purpose: This feature is about the intended purpose to explore the software or focus on qualitative aspects.
  - Randomization has a exploratory purpose and is useful to map a bigger picture of the software as far as behaviour is concerned.
  - The purpose of scenarios is qualitative, and thus can be effective in generating models local to the scenario.

Since, both the choices have their fair share of advantages, both of those choices were considered in this work which gave a deeper aspect to the study. Those aspects are discussed in Chapter 6. The method for randomization is as follows.

API functions were indexed, i.e., a number was associated with every function. A sequence of random numbers was generated. The sequence included numbers between 0 and maximum value, where maximum value was equal to total number of functions in the API. The length of this sequence is variable. The functions were executed if their index value matched the number in the random sequence. Thus, every time a new random sequence was generated, the execution of
functions depended on the sequence of random numbers which corresponded to the indices of those functions. The indices of the bounded queue API are mentioned in Chapter 6.

At first randomization was experimented with in stage 1 of development, but was dropped because of concerns related to it, these concerns are discussed in Chapter 6. Since, this work was directed towards embedded software, RTOS being a state-full system, random input and scenario generation for the existing internal states would have been too large [7].

![Diff tool taking a diff of two versions.](image)

Figure 9: Diffuse tool taking a diff of two versions.

In 2nd stage of development, scenarios were considered for trace generation. Since, the end goal was to compare behavioural models, changes in the source code had to be targeted. These changes had to be functional in nature. Also, only those functions which were allowed to be called by the user were considered for evaluation using a diff tool.

For this purpose, diffuse an open source diff tool was used, which could highlight the changes in the source code [19]. It highlighted all the changes which were syntactic as well as functional. An instance of diffuse in action is shown in Figure 9. None the less, it was an effective method to take the diff of the source code from version to version for the files that had methods which were allowed to be called by the user, and then design the scenario around the methods that changed functionally. Kernel space functions were not a part of this process. Checking if the methods changed functionally was done manually.
Thus to compare behavioural models, scenarios that included functionally changed methods were designed along with general scenarios.

4.4 Scenarios used in this work

4.4.1 RTOS functions used for the Scenarios

The user space functions which are instrumented for the scenarios are described below. Detailed specification is provided in Chapter 9.

1. init_kernel()
   This function initializes the kernel and its data structures and leaves the kernel in start-up mode. The init_kernel call must be made before any other call is made to the kernel.

2. create_task(void(*task_body)(), uint deadline)
   This function creates a task. If the call is made in startup mode, i.e., the kernel is not running, only the necessary data structures will be created. However, if the call is made in running mode, it will lead to a rescheduling and possibly a context switch.

3. run()
   This function starts the kernel and thus the system of created tasks. Since, the call will start the kernel it will leave control to the task with tightest deadline. Therefore, it must be placed last in the application initialization code. After this call the system will be in running mode.

4. terminate()
   This call will terminate the running task. All data structures for the task will be removed. Thereafter, another task will be scheduled for execution.

5. create_mailbox(int nof_msg, int size_of_msg)
   This call will create a Mailbox. The Mailbox is a FIFO communication structure used for asynchronous and synchronous communication between tasks.

6. send_wait( Mailbox* mBox, void* Data)()
   This call will send a message to the specified mailbox.

7. receive_wait( Mailbox* mBox, void* Data)
   This call will attempt to receive a message from the specified mailbox.
4.4.2 Communication Scenarios

1. Initialize the RTOS kernel, create a mailbox of size 1, send a message using `send_wait()` and receive it using `receive_wait()`.

2. Initialize the RTOS kernel, create a mailbox of size 5, send a message using `send_no_wait()` and receive it using `receive_no_wait()`.

3. Initialize the RTOS kernel, create a mailbox of size 5, receive a message using `receive_wait()`, then send a message using `send_wait()`, then do the opposite.

4. Initialize the RTOS kernel, create a mailbox of size 5, continuously send a message using `send_no_wait()` even after the message box is full then continuously receive it using `receive_no_wait()` even after the message box is empty.

5. Initialize the RTOS kernel, create a mailbox of size 1, send a message using `send_wait()` and wait until deadline.

6. Initialize the RTOS kernel, create a mailbox of size 1, receive it using `receive_wait()` and wait until deadline.

7. Initialize the RTOS kernel, create a mailbox of size 1, receive it using `receive_no_wait()` and then send a message using `send_no_wait()`.

4.4.3 Task Scenarios

1. Initialize the RTOS kernel, create a task of deadline 100, terminate the task.

2. Initialize the RTOS kernel, create the first task of deadline 100, create a second task of deadline 90 from inside the first task, create a third task of deadline 80 from inside the second task, create a fourth task of deadline 70 from inside the third task. Start terminating the task in the reverse order from fourth to first.

4.4.4 Timing Scenarios

Timing scenarios differ from task scenarios since timing scenarios contain the `wait()` method which blocks the tasks for a certain time.

1. Initialize the RTOS kernel, create a task of deadline 100, make it wait and then terminate the task.

2. Initialize the RTOS kernel, create the first task of deadline 100, create a second task of deadline 70 from inside the first task.
Make the task 70 wait for 20 time units, then terminate it. Make the first task wait for 70 time units and then terminate it as well.

3. Initialize the RTOS kernel, create the first task of deadline 100, create a second task of deadline 70 from inside the first task. Make the task 70 wait until deadline, then terminate it. Make the first task wait until deadline and then terminate it as well.

4.5 **ANALYSIS OF THE TRACES**

After the traces are generated, the tool needs to analyse them in-order to convert those traces into states and transitions.

4.5.1 *Choices related to analysis of traces*

<table>
<thead>
<tr>
<th>Feature</th>
<th>Machine Learning</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Pattern</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Ease of implementation</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5: This table depicts the choice of abstraction for analysis of the traces by the tool.

As shown in Table 4, the following features were noted among the choices of Abstraction and Machine Learning to perform analysis of traces.

- **Portability**: This feature compares if the method shall make the tool applicable to other software or be confined to the one designed for.
  - Machine Learning has the advantage of portability, whereas Abstraction is a hard-coded technique and gets tied to the software it is designed for [16, 8].

- **Pattern**: This feature compares the methods based on a need for pattern in the traces.
  - Having a pattern in the data is crucial for Machine Learning classifiers to be effective, whereas Abstraction has the advantage even if there is no pattern in the data [8], which is not the case generally when traces are generated, but its a point in favour of abstraction.

- **Ease of Implementation**: This feature compares how feasible the method is to implement at a early stage of development.
– Machine Learning is a high level approach to consider when building a tool from scratch, where as abstraction although tedious to adapt to a new software, is simpler to implement as it resembles a fixed decision tree.

![Figure 10](image)

Training of Machine learning classifiers requires large amount of trace data, which in turn requires large number of different scenarios. With randomization it is possible to generate large amount of traces automatically, but RTOS being an embedded software, needs additional logic to call the different methods. This complicates trace generation process. Also, trace data that is obtained from scenarios due to instrumentation has many variations as shown in Figure 10.

Training a classifier on such a data set is a daunting task, as it may lead to mis-classifications. The tool is in its nascent stages, using an advanced technique such as machine learning would have side tracked the study, since the goal was to visually compare behavioral models.

Thus, abstraction process was used, as it can model the behavior as a state machine based on observed values, staying true to the intention of behaviour modelling of embedded software using execution traces. Hence, customizing the decision tree of abstraction process to suit the data was simple to implement, rather than using machine learning or any other learning algorithm on traces that had variation in data related to every method.

As a part of future work, integrating the technique of machine learning into SMT 2.0 to learn a behavioral model is discussed in Chapter 6.

4.5.2 Abstraction

In Figure 11, an example of complex.state.list is illustrated. pre.state list contains data before execution of enqueue() method. post.state
list contains data after the execution of `enqueue()` method. As illustrated in Figure 11, the `pre_state` and `post_state` lists store data from `isEmpty()`, `size()` and `isFull()` methods. These methods are executed before and after the `enqueue()` method as a part of instrumentation.

Thus when `pre_state` and `post_state` list is combined with the method being instrumented, in this case, the `enqueue()` method, it results in a `complex_state_list`. So `complex_state_list` is appended for every method that is executed and is instrumented.

Thus `complex_state_list` contains lists of all the functions that are executed along with the `pre_state` and `post_state` lists.

Abstraction is the process of analyzing the raw data contained in the `pre_state` and `post_state` list associated with every executed method. Thus, when `complex_state_list` undergoes the process of abstraction, `simple_state_list` is obtained.

Figure 12, illustrates a simple example of abstraction for the `enqueue()` method.

In Figure 12, line marked 1 illustrates the functions used for instrumenting the `enqueue()` function. In the same Figure 12, the line marked 2 illustrates one of the lists in the `complex_state_list`. It
Figure 13: This figure depicts the output of abstraction process which is a simple_state_list

\[
\text{Simple_state_list} = [\text{Empty}, \text{enqueue}(5), \text{NotEmpty}] \\
\]

also shows data collected as a result of instrumenting the enqueue() function with isEmpty(), size() and isFull() functions. Thus line 1 and 2 in Figure 12 show the co-relation between instrumentation and data collected. Line 3 in Figure 12 is the result of abstraction, which is obtained after the data in line 2 is classified according to certain criteria.

So to the left of enqueue() method on line 3 in Figure 12, the state is classified as \textbf{Empty} state. Since the data associated with isEmpty method is equal to 1. To the right of enqueue() method on line 3 in Figure 12, the state is classified as \textbf{NotEmpty} state. Since the data associated with size method is equal to 1. This is how abstraction process analyzes data contained in the pre_state and post_state lists associated with the executed method.

The states that have been classified are referred to as abstract states. The process which maps the raw data to abstract states is called abstraction. simple_state_list as shown in Figure 13 is thus a collection of lists that have undergone the abstraction process.

The code responsible for performing classification is a fixed decision tree, which is developed by observing data values and is called as Abstraction_engine().

4.6 Merging Algorithm

After the traces have been analysed a Prefix Tree Acceptor (PTA) is generated. PTA is a tree data structure annotated at the edges by transitions. Common edges in a PTA share the same path up-to a point where they diverge. Once this PTA is generated, it has to be merged to combine equivalent states and obtain a compact leaned behavioural model.

Four merging algorithms were studied in this work and their properties analysed for implementation purpose. The four types of merging algorithms are described in the following subsections.
4.6.1  **GK-tails**

GK-tail produces an EFSM by processing a set of interaction traces. It works in four main steps as mentioned in [10]:

1. Merge traces.
2. Generate predicates associated with traces.
3. Create an initial EFSM.
4. Merge equivalent states to obtain the final EFSM.

Merging of the states is done by a modified K-tails algorithm. During merging the pair of states to be merged is decided by the choice of merging criterion. This merge criterion is where GK-tails differs from the original K-tails algorithm. These criterion consider the path between the two states to be merged and the predicates of those paths. Predicates are a general descriptions of the conditions related to variables and their data, which are captured in a \{var, data\} format. The merging criterion used for GK-tails merge algorithm are as follows [10]:

1. Equivalency criterion: If path between the chosen pair of states match and the predicates match as well, then the selected states are equivalent and ready to merge.
2. Weak subsumption: If the path between the chosen pair of states match, but the predicates of the paths of one state are a subset of the other, then this criterion is called weak subsumption.
3. Strong subsumption: If the path between the chosen pair of states and predicates of those paths of one state are a subset of the other, then this criterion is called strong subsumption.

4.6.2  **EFSM**

The EFSM inference algorithm is a modified version of the Evidence Driven State Merging (EDSM) algorithm. The inclusion of machine learning classifiers makes EFSM different from EDSM. As shown in Algorithmus 1, the EFSM inference algorithm and its functionality is as follows [16].

- The algorithm starts with a `prepareDataTraces()` method, which groups the traces function wise along with data and a class variable to each data point, which denotes the label of the following function.
Algorithm 1: EFSM inference algorithm

Data: EFSM, $\Delta$, $k$, $c$, DataTraces, $s_1$, $s_2$, $t_1$, $t_2$, Vars

/* Here $A$ is shorthand for the collection ($S$, $s_0$, $F$, $L$, $\Delta$, $T$) as per definition of an EFSM. Components of $A$ are denoted by subscript (e.g. $A_s$) */
/* $s_1$, $s_2 \in S$ */
/* $t_1$, $t_2 \in T$ */
/* Given a variable domain $V$ and a set of classes $C$, a DataTrace $T_D$ of size $n$ can be defined as a set [{($v_0,c_0$),...,$(v_n,c_n)$}], where a vector of variable assignments $v_i$ is mapped to its corresponding class $c_i$ */
/* Vars is a one-to-many mapping from transitions in $T$ to trace elements in Traces. $k$ is an (optional) integer $\geq 0$ representing a minimum merge score */

Result: An EFSM consistent with Traces.

infer(Traces,$k$) begin:

DataTraces ← prepareDataTraces(Traces)
$\Delta$ ← inferClassifiers(DataTraces)
($A$,Vars) ← generatePTA(Traces,$\Delta$)
while ($s_1$, $s_2$) ← choosePairs($A$, $\Delta$, $k$) do

($A'$,Vars') ← merge($A$, ($s_1$, $s_2$), Vars)
if consistent($A'$,$\Delta$,Vars) then

$A$ ← $A'$
Vars ← Vars'
end

end

return ($A$)

end

merge($A$, $s_1$, $s_2$, Vars) begin:

$A_S$← $A_S$ \ $s_1$;
$A_F$← $A_F$ \ $s_1$;
$A_T$← changeSources($s_1$out, $s_2$, $T$);
$A_T$← changeDestinations($s_1$in, $s_2$, $T$);
while ($t_1$, $t_2$) ← equivalentTransitions($A_T$, $s_2$, $A_\Delta$) do

if ($t_1$dest == $t_2$dest) then

Vars($t_2$) ← Vars($t_2$) $\cup$ Vars($t_1$)
$A_T$ ← $A_T$ \ $t_1$
else

($A$,Vars) ← merge($A$, ($t_1$dest, $t_2$dest), Vars);
end

end

return ($A$, $\Delta$)

end
• These data traces are processed by a classifier and is available as denoted by $\delta$.

• The method `generatePTA()` generates a PTA.

• The `choosepairs()` method chooses two states to be merged which are equivalent. States are proclaimed as equivalent if their attached data values predict the same label, thus integrating the machine learning classifiers which makes it different from EDSM.

• The merge function primarily relies on the `equivalentTransitions()` method. It proclaims two transitions as equivalent based on both, label as well as data. The merged machine $A'$ and data $\text{Vars}$ are checked for consistency with classifier $\delta$.

The end result of EFSM inference algorithm is an EFSM consistent with Traces.
4.6.3 MSM Algorithm

**Algorithmus 2**: MSM algorithm for state merging based on state coloring and state labeling

**Input**: A non-empty initial positive and negative sample $(S^+, S^-)$

**Input**: Labeling and coloring constraints

**Output**: A DFA $A$ consistent with $(S^+, S^-)$ and all constraints

1. // Compute a PTA, let N denote the number of its states
2. // PTA ← Initialize$(S^+, S^-); \pi \leftarrow \{0\}, \{1\}, ..., \{N - 1\}$
3. // Merge all states according to labeling constraints
   \[ \text{while } (B_i, B_j) \leftarrow \text{FindSameBlocks}(\pi) \text{ do} \]
   \[ \pi \leftarrow \text{Merge}(\pi, B_k, B_l) \]
4. end
5. // Main state-merging loop
6. while $(B_i, B_j) \leftarrow \text{ChoosePair}(\pi)$ do
7. try
8. \[ \pi \leftarrow \text{Merge}(\pi, B_i, B_j) \]
9. end
10. catch
11. // next state pair to consider
12. end
13. return $(\text{PTA}\backslash A)$
14. end
15. // This function merges two blocks and removes non-determinism
16. // Recursively while checking coloring constraints
17. \text{merge}(\pi, B_i, B_j) \text{ begin:} \]
18. \[ \text{if } (\text{incompatible}(B_i, B_j)) \text{ then} \]
19. \[ \text{raise avoid} \]
20. end
21. \[ \text{while } (B_k, B_l) \leftarrow \text{FindNonDeterminism}(\pi, B_i, B_j) \text{ do} \]
22. \[ \pi \leftarrow \text{Merge}(\pi, B_k, B_l) \]
23. end
24. return $\pi$
25. end

MSM is a state merging algorithm Algorithmus 2 that generates a DFA using mandatory merge constraints. It not only takes into consideration the states which are not to be merged, but also considers the states that can be merged as well. For this purpose, MSM uses coloring constraints and labeling constraints. Coloring constraints contain the knowledge about incompatibilities between states of a PTA. Labeling constraints contain the information of the states with same label that can be merged. The algorithm also uses positive and negative
samples of traces. Positive samples are the traces that can be executed by the software. Negative samples are the traces that are not allowed for execution by the software [17].

MSM algorithm starts by generating a PTA from positive and negative samples using the Initialize() method. The FindSameBlocks() methods provides the states with same label to be merged using the labeling constraints. Thus, states with labeling constraints are merged before the main merge loop [17].

The main merge loop starts with ChoosePair() method, which chooses a set of state pair based on standard lexicographic order of strings. The merge method recursively merges the states to remove any non-determinism. It checks for coloring constraints only as labeling constraints have already being satisfied. On the first occurrence of a DFA, the merging process is stopped [17].

4.6.4 Choices related to merging algorithm

<table>
<thead>
<tr>
<th>Feature</th>
<th>GK-tails</th>
<th>EFSM inference</th>
<th>MSM</th>
<th>Custom merge algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Check label and data</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Choose pairs based on predictions</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State labelling for compatibility</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>State coloring for incompatibility</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: This table depicts the choice of merging algorithm and the features selected for the custom merge algorithm.

As shown in Table 6 the following features were noted among the choices of GK-tails, EFSM inference algorithm, MSM algorithm.

- **PTA**: PTA is an initial automaton that is generated from traces.
  - All the tools generate an initial PTA from input traces with subtle variation in methods.

- **Check Label and Data**: This is a feature used during merging process to identify common transitions and data associated with those transitions.
  - GK-tails and EFSM considered the transitions and data associated with them while merging two states.
  - MSM had domain knowledge factored in for merging.

- **Choose pairs based on prediction**: A feature to choose states as merge candidates based on predictions from classifiers.
  - Only EFSM used this feature, and claims to be better than GK-tails and Adabu [16].
– Other tools did not use machine learning classifiers to guide their merging process.

• State labeling for compatibility: A feature to consider a state fit for merging with a state with same label.
  – This feature was used by MSM algorithm as mandatory merge constraints, that need to be merged first.
  – Other tools focused more on incompatibility of states.

• State coloring for incompatibility: A feature to signify its incompatibility to merge.
  – MSM used this feature in addition to state labeling.
  – Other tools find incompatibility of states based on non-common transitions or classifiers, but do not mention state coloring.

State information was already obtained by abstraction process. It would be logical to take advantage of that information. Thus, the concept of state labeling was important, which was used by MSM algorithm [17]. The abstracted traces if used to populated a graph data structure would simplify the input to the merging algorithm, than generating a PTA. A non-recursive approach would further simplify the process of merging. But, while merging transition as well as data has to be considered. These were the requirement which had to be met in-order to simplify the implementation of merging process.

The algorithms studied in Table 6, satisfied one or two requirements, but not all requirements individually. Thus, there was a need to develop a custom algorithm, which would satisfy those requirements to simplify the implementation of merging process.
Algorithmus 3: Custom Algorithm for merging the states of initial graph.

Input: Object of class Automaton, denoted by $A$
Input: simple_state_list which is obtained after abstraction.
Output: An Automaton which represents the behavioural model

```plaintext
// $A_1 \leftarrow$ create_initial_graph($A$, simple_state_list).

main_merge($A_1$) begin:
  $A_{\sigma} \leftarrow$ start_merge($A_1$)
  $A_M \leftarrow$ filter_efsm($A_{\sigma}$)
  $A_G \leftarrow$ make_final_efsm($A_M$)
  return $A_G$
end

start_merge($A_1$) begin:
  $e =$ find_all_equivalent_states($A_1$)
  for $l \in e$ do
    if (len($l$) $<$ 0) then
      // cannot merge empty list
    else if (len($l$) $=$ 1) then
      node $\leftarrow l_0$
      tnl $\leftarrow$ all_neighbours_of_node(node) $\in A_1$
      $A_{\sigma} \leftarrow$ merge_given_list(tnl)
    else
      $A_{\sigma} \leftarrow$ merge_given_list($l$)
    end
  end
  return $A_{\sigma}$
end

merge_given_list($l$) begin:
  $n_s \leftarrow l_0$
  for $n_d \in l$ do
    if attribute($n_s$) $==$ attribute($n_d$) then
      if labels match then
        if data match then
          // Add transition and data to $\sigma$ during first run
        else
          // Add new data to $\sigma$
        end
      else
        // Add new transition along with data to $\sigma$
      end
    else
      // mark in incompatible and do not merge
    end
  end
  return $\sigma$
end
```
The custom merge algorithm starts by building an initial graph from the `create_initial_graph()` method.

The main merge loop then starts by processing this initial graph \( A_I \) using the `start_merge()` method, which generates a intermediate transition list \( A_I \). This intermediate transition list is then processed to remove redundancy in state pairs using `filter_efsm()` method. The filtered transition list \( A_M \) is then displayed with `make_final_efsm()` method.

In this algorithm, equivalent states are the states that have the same attribute, much like state labeling concept of MSM algorithm \([17]\). The `start_merge()` method starts by finding all equivalent states using initial graph \( A_I \) as input, and stores them in a list. All those lists are collectively stored in \( e \). It then loops over \( e \), with \( l \) being the list that consists all the equivalent states. The length of the list is checked to identify if there exists a single node with unique attribute. Merging the neighbours of this node is done by generating \( tnl \) and giving it to the `merge_given_list()` method. If length of \( l \) is greater than one, then the list is given directly to `merge_given_list()` method.

The `merge_given_list()` method merges the equivalent states of input list \( l \). The node marked \( n_S \) is the source node. The node marked as \( n_d \) is the destination node. Attributes of \( n_s \) and \( n_d \) are checked. If attributes are equivalent then a second check is performed for matching labels. If labels match then their data is considered, if the data is different, then new data is added to \( \sigma \), else, just the transition is added to \( \sigma \) corresponding to \((n_S, n_d)\). During the first run of the loop, transition and data needs to be added. If labels do not match, then the new transition along with its data is added to \( \sigma \) corresponding to \((n_S, n_d)\). If the attributes do not match, then the states are considered incompatible. `merge_given_list()` returns the intermediate node transition map \( \sigma \).
Algorithmus 4: Custom Algorithm for filtering and displaying the final efsm

**Input**: Object of class Automaton, denoted by \( A \)

**Input**: intermediate transition list \( \sigma \) obtained from start_merge().

**Output**: An Automaton which represents the behavioural model

\[// A_I \leftarrow create_initial_graph(A, simple_state_list)\]

\[filter_efsm(A, \sigma) \begin{align*}
trans_I & \leftarrow get_transition_list \in A_{\sigma} \\
sl & \leftarrow state_list \in A \\
pl & \leftarrow generate_pair_list(sl) \\
& \text{for } p \in pl \text{ do} \\
& \text{for } k \in trans_I \text{ do} \\
& \text{if } (\text{attribute}(p) == \text{attribute}(k)) \text{ then} \\
& \text{if labels match then} \\
& \text{if data match then} \\
& \quad // Add transition and data to to M if first run \\
& \text{else} \\
& \quad // add the new data to the label \\
& \quad // for the current pair p and store in M \\
& \text{end} \\
& \text{else} \\
& \quad // add the new transition for the current pair p \\
& \quad // and store in M \\
& \text{end} \\
& \text{end} \\
& \text{return } M \\
\end{align*}\]

\[make_final_efsm(A, M) \begin{align*}
G & \in A // Populates a graph G with the states and transitions along with \\
& // the data from M, assign a color to the states. G \leftarrow M \in A \\
& \text{return } A_G \\
\end{align*}\]

\(\text{filter_efsm()}\) takes \( \sigma \) as the input and stores it in \( trans_I \). It then generates a list \( pl \) consisting of all possible state pairs. The state pairs that are common to \( pl \) and \( trans_I \) and match in attribute are considered for processing only. These state pairs are checked for labels and data. If label and data match, add to \( p \) if its a first run. If labels match but data is different then add the new data to the label for the current pair \( p \) and store in \( M \). If label and data both are different then add the new transition for the current pair \( p \) and store in \( M \). The final node transition map \( M \) is returned by \(\text{filter_efsm()}\).

\(M\) is then used to populate a graph \( G \) by \(\text{make_final_efsm()}\) and displayed using \(\text{Draw}(A_G)\).
4.8 VISUAL COMPARISON

If two versions of a software are compared directly by taking a difference of the source codes then all kinds of changes are highlighted as shown in Figure 9. All kinds of changes refer to syntactic changes as well as functional changes. Syntactic changes being, changes in structure of the code considering the delimiting brackets, changes in variable names, changes in function names and so on. Functional changes are also highlighted. This creates a lot of ambiguity to extract pure functional changes which help in identifying the behavioral changes between the two versions [6].

Differential Symbolic Execution [6] is a technique solving this problem using symbolic execution. But this technique precisely identifies the behavioral difference of the methods between version and not their interaction with other functions. Differential symbolic execution is a powerful technique and its usage to further improve this work is discussed in Chapter 6.

Another way to solve this problem is to obtain behavioral models of the two versions and visually compare those models to find differences. The advantage being, direct functional changes can be compared directly by taking the difference of the two behavioral models and not their source codes. Thus a visual comparison algorithm Algorithm 5 was developed to identify difference in behaviour among the versions of a software. Visual comparison algorithm and the idea to visually compare two behavioral models is the unique contribution of this work. Difference of source code was taken for kernel space functions after certain behavioral models were obtained and visually compared. Since, the kernel space functions of the RTOS were not instrumented. To pin point the method responsible for indirectly influencing the behavior observed in behavioral model, visual comparison was used along with source code difference.
Algorithmus 5 : Algorithm for comparison of two learned behavioural models

**Input** : Object of class Automaton, denoted by $X$

**Input** : Object of class Automaton, denoted by $Y$

**Output** : Objects of class Automaton containing information of different transitions

```plaintext
compare_model($X$, $Y$) begin:

    $x := 0$, $y := 0$
    $cd_x$, $fd_x$ ← $M \in X$
    $cd_y$, $fd_y$ ← $M \in Y$

    for $k_x \in fd_x$ do
        for $k_y \in fd_y$ do
            if (attribute($k_x$) == attribute($k_y$)) then
                $l_x$ ← $k_x \in fd_x$, $l_y$ ← $k_y \in fd_y$
                //Match $l_x$ with $l_y$
                if Matched then
                    $x ← 1$, $y ← 1$
                else
                    $x ← 0$, $y ← 0$
                end
            end
        end
    end

    if $cd_x == 0$ and $cd_y == 0$ then
        //core dict is empty, print efsm matched
    else
        //print efsm not matched
    end

    return $cd_x$, $cd_y$
end

visualize_final_efsm($X$) begin:

    $M \in X$, $cd_x \in X$
    //Populate a graph $G \in X$ with the nodes and transitions in $M$.
    // color the graph $G$ with the transitions and respective data associated with the transitions
    // using the information in $cd_x$
    $X_G$ ← $G \in X$

    return $X_G$
end
```
The visual comparison algorithm 5 takes two objects X and Y of class automaton as inputs. cdₓ and fdₓ contain the final state pairs and their corresponding transitions for automaton X. cdᵧ and fdᵧ contain the final state pairs and their corresponding transitions for automaton Y.

The equivalency of state pairs Kₓ and Kᵧ is checked by comparing their attributes. If the attributes match, then the lists lₓ and lᵧ containing the transitions for that particular state pair belonging to X and Y, respectively, are matched. If lₓ and lᵧ match, x and y are 1, then state pairs Kₓ and Kᵧ and their corresponding transitions are deleted from cdₓ and cdᵧ. If lₓ and lᵧ do not match, then transitions common to them are removed. lₓ and lᵧ are reassigned to their respective state pair Kₓ and Kᵧ in cdₓ and cdᵧ. When cdₓ and cdᵧ both are empty, the EFSM X and Y are deemed as matched, else otherwise. compare_model() returns cdₓ and cdᵧ.

To visualize the changes between X and Y, visualize_efsm() method is used. It populates a graph G with the nodes and transitions in M. It colors the transitions using the information in cdₓ. Thus, making it possible to visually justify if one model differs from another. The populated and colored graph G is displayed using Draw(XG).

4.9 Development Stages of the Tool

Figure 14: Development stages of the tool

Figure 14 shows the order in which the development took place.

- The development was started with SMT 1.0, which was directed towards obtaining a behavioral model of a bounded queue. This was the first stage of development as shown in Figure 14.
• The second stage was related to the development of SMT 2.0 which was also directed towards obtaining a behavioral model of a bounded queue as shown in Figure 14, but with major changes as compared to first stage.

• The third stage was related to the adaptation of SMT 2.0 to a RTOS kernel as the embedded software to be modelled for its behavior. The methods were directed to obtain a model for the RTOS kernel.

4.10 SUMMARY OF CHOICES

<table>
<thead>
<tr>
<th>Category</th>
<th>Method</th>
<th>FSMGen</th>
<th>Adabu</th>
<th>AGSBM</th>
<th>EFSM tool</th>
<th>SMT 1.0</th>
<th>SMT 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Traces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technique</td>
<td>Symbolic execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace generation</td>
<td>Scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace generation</td>
<td>Randomization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traces</td>
<td>Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traces</td>
<td>Machine Learning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>Modified K-tails</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>EDSM with ML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>MSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging</td>
<td>Custom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Model type</td>
<td>FSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Model type</td>
<td>EFSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparison</td>
<td>Visual Comparison</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7: This table illustrates the choices from methodology that were made for the tools SMT 1.0 and SMT 2.0, green signifies derived from other tools, grey signifies a combination, yellow signifies unique to that tool.

The decision to use a particular method for the specific categories culminated in two tools with unique characteristics.

The decisions taken during the first stage were as follows:

• The tool was chosen to operate on traces as input.

• Random execution of functions was experimented with in this stage for generation of traces based on instrumentation.

• Abstraction of traces was decided as a method to convert traces into states and transitions.

• The model was decided to be represented as a FSM.

These decisions culminated in SMT 1.0
The decisions taken during the second stage were as follows:

- The tool was chosen to operate on traces as input.
- Scenarios were designed manually to generated traces based on instrumentation.
- Abstraction of traces was decided as a method to convert traces into states and transitions.
- Development of a custom merge algorithm was decided for this tool.
- The model was decided to be represented as a EFSM.
- Visual compare option was added to this tool.

These decisions culminated in SMT 2.0. The decision to include visual comparison option helped in the third stage of the tool for comparing the behavioral models of RTOS kernels.
RESULTS

This chapter is divided into two parts. First part is about the tools that were developed. Second part is about the results that were obtained by applying the tool on a RTOS kernel.

5.1 SMT 1.0

Figure 15: This figure is about the first stage of the development stage where SMT 1.0 was developed with bounded queue as the software to be modelled.

SMT 1.0 was the result of decisions taken during first stage of development as signified by the highlighted green box in Figure 15. SMT 1.0 randomly executed functions of the bounded queue. Data collected from these executions was processed by SMT 1.0 using a abstraction engine. Final output was a behavioral model of the queue. Following section describes how SMT 1.0 obtained a behavioral model of the bounded queue. Description of bounded queue API is provided in Chapter 8.
import networkx as nx
from nxpd import draw

q = Queue(max_size)

seq = make_random_sequence()

complex_state_list = generate_complex_state_list(seq,q)

simple_state_list = convert_to_abstract_state(
    complex_state_list,max_size)

initial_clean_state_machine = clean_up_state_machine(
    simple_state_list)

G = populate_graph(initial_clean_state_machine)
draw(G)

teacher_result = "fail"

final_clean_state_machine = initial_clean_state_machine
new_complex_state_list = complex_state_list

while True:
    teacher_result, new_seq = teacher_analysis(
        final_clean_state_machine,max_size)
    count = count + 1
    if teacher_result == "pass":
        print_table(final_clean_state_machine)
        break
    new_complex_state_list = update_complex_state_list(
        new_seq,new_complex_state_list,q)
    new_simple_state_list = convert_to_abstract_state(
        new_complex_state_list,max_size)
    final_clean_state_machine = clean_up_state_machine(
        new_simple_state_list)

FG = populate_graph(final_clean_state_machine)
draw(FG)

Listing 2: Order of SMT 1.0 function calls
Listing 2 shows the steps taken by SMT 1.0 to obtain a behavioral model which are described as follows.

5.1.1 Step 1: Generate a random sequence

The method `make_random_sequence()` is used to generate a random sequence. An example of one such random sequence is [0, 0, 4, 4].

5.1.2 Step 2: Instrumentation of functions

![Complex State List](image)

Figure 16: The output for `generate_complex_state_list()` function of SMT 1.0

The method `generate_complex_state_list()` accepts the random sequence generated in step 1 as an input. It then executes the functions according to their index positions and generates a `complex_state_list`. This `complex_state_list` is generated by the method of instrumentation described in Chapter 4. Figure 16 shows a `complex_state_list` which is the output of `generate_complex_state_list()` for the random sequence [0, 0, 4, 4].

5.1.3 Step 3: Abstract the states

![Abstract States](image)

Figure 17: The output of the `convert_to_abstract_state()` function of SMT 1.0

The `complex_state_list` is given as a input to `convert_to_abstract_state()` which uses the process of abstraction described in Chapter 4 to generate a `simple_state_list`. Figure 17, shows an example of `simple_state_list` obtained after the abstraction process.

5.1.4 Step 4: Initial behavioral model

The `simple_state_list` is processed to remove any duplicate transitions which may arise due to repeated execution of the same function, since random execution is employed in this tool.
The `clean_up_state_machine(simple_state_list)` method removes the duplicate transitions and gives the output as an `initial_clean_state_machine`.

At this step, `initial_clean_state_machine` obtained is displayed as an initial behavioral model by populating a graph data structure using `populate_graph(simple_state_list)`. Figure 18, shows the initial behavioral model.

![Graph](image)

**Figure 18: The initial model obtained by SMT 1.0**

5.1.5 **Step 4: Final behavioral model**

The initial behavioral model is then used by `teacher_analysis()` method to build a better model of the queue by repeating the steps 1 to 3. The method `teacher_analysis()` checks if the length of the list containing the abstracted transitions i.e., `final_clean_state_machine` is equal to the estimated transitions.

The estimated transitions for a particular software are number of states multiplied by the number of functions. Assuming that all the functions can be called in all the states, and the number of states are known.

Once the length of the `final_clean_state_machine` is equal to the estimated transitions, the process is stopped and the final behavioral model is displayed using the `draw()` function.

Since, the estimated transitions may not be accurate, the behavioral model is not a complete model of the queue.

If pre-conditions are considered it may be possible to obtain precise estimated transitions, but at the cost of accuracy of the behavioral model, as transitions that are illegal in some states may not be a part of the final behavioral model. Even if estimated transitions are precise, the queue has to be tried for different sizes including negative,
zero and one. Thus, tending towards a complete behavioral model. But, since the purpose of the model was representing behavior, and not classifying correct or incorrect behavior, the inaccuracies were tolerated [16].

Figure 19, shows the final model generated by SMT 1.0.
5.2 SMT 2.0

Figure 20: This figure is about the second stage of the development stage where SMT 2.0 was developed with bounded queue as the software to be modelled.

SMT 2.0 was the result of decisions taken during second stage of development as signified by the highlighted blue box in Figure 20.

```python
from nxpd import draw
from XDFA import automaton
from merging_basic_efsm import *
from trace_reader import *

trace_object = open('tracefile.txt', 'r')
complex_state_list = read_trace(trace_object)
simple_state_list = abstraction_engine(complex_state_list)
dfa = automaton()
initial_graph = create_initial_graph(dfa, simple_state_list)
draw(initial_graph.G)
merged_automaton = start_merge(initial_graph)
draw(merged_automaton.G)
model = filter_efsm(merged_automaton)
make_final_efsm(model)
draw(model.final_efsm)
```

Listing 3: Order of SMT 2.0 function calls
Listing 3, shows the steps taken by SMT 2.0 to obtain a behavioral model which are described as follows.

5.2.1 **Step 1: Instrument the functions**

```
trace
0,0,0,Queue_maxSize_2,True,0,False
True,0,False,enqueue,False,1,False
False,1,False,enqueue,False,2,True
False,2,True,dequeue,False,2,True
trace
0,0,0,Queue_maxSize_2,True,0,False
True,0,False,enqueue,False,1,False
False,1,False,enqueue,False,2,True
False,2,True,dequeue,False,2,True
```

Figure 21: This figure shows the trace logged for the bounded queue scenario after instrumentation by SMT 2.0.

Figure 21, shows the generated traces which is the result of instrumentation and use of scenarios, since the decision was taken to give traces as input to the tool, as mentioned in Chapter 4. These traces are stored in a text file.

5.2.2 **Step 2: Read the trace**

The method `read_trace(trace_object)` is used to read `tracefile.txt` and its output is a `complex_state_list` which is similar to Figure 16.

5.2.3 **Step 3: Abstract the states**

The `complex_state_list` from step 2 is given as an input to `abstraction_engine(complex_state_list)` method, which generates a `simple_state_list` using the method of abstraction described in Chapter 4. The output is similar to Figure 17.

5.2.4 **Step 4: Creating an initial graph**

In this step, an object of class automaton is created, which is named `dfa`, as mentioned in Listing 3.

As shown in Figure 22 an initial graph is created by the `create_initial_graph()` method which takes the object `dfa` and `simple_state_list` as input.
5.2.5 Step 5: Merging the initial graph

The initial graph is then merged using a custom merge algorithm and the resulting output is shown in Figure 23.

5.2.6 Step 6: Filtering the merged state machine

During the merging process the transitions of the initial graph are not deleted and hence these old transitions must be filtered out in order to get the actual merged graph. This is done using the filter_efsm() method.
5.2 SMT 2.0

5.2.7 **Step 7: Displaying the final behavioral model**

Figure 23: This figure shows the intermediate merged graph generated by the `start_merge()` function of SMT 2.0

Figure 24: This figure shows the final behavioral model obtained from `make_final_efsm()` method and displayed by the `draw()` method.

`make_final_efsm()` method is then used to decorate a state machine to represent the final behavioral model shown in Figure 24. This final behavioral model is displayed using the `draw()` method.
Figure 25: This figure shows behavioral models of version 1 and version 2 of the bounded queue, they are highlighted with transitions that are distinct after comparing the two models using visual comparison algorithm.

5.2.8 Comparison of behavioral models

For the purpose of comparison, two versions of the bounded queue code were produced. Version 1 was injected with an error related to the dequeue() function and version 2 of the queue was devoid of injected errors.

The scenario was to add two elements in the queue, and remove one element from it. Thus, the queue was initialized, enqueue() was called twice followed by a call to dequeue() method.

The traces were collected and processed according to the steps 1 to 6. The behavioral models of scenarios for both versions were given as input to the compare function model_compare(model1, model2). model_compare() method analysed the two models for their different transitions, this analysis is then highlighted with colour by visualize_final_efsm() method. Comparison was performed according to the visual comparison Algorithmus 5.

Figure 25 shows behavioral models of version 1 and version 2 of the bounded queue. They are highlighted with transitions that are different after comparing the two models using visual comparison algorithm.
5.3 ADAPTING SMT 2.0 TO THE RTOS KERNEL

Figure 26: This figure is about the third stage of the development stage where SMT 2.0 was adapted to a RTOS kernel, which was the software to be modelled.

Adaptation of SMT 2.0 to a RTOS kernel is done in the third stage of development. It was the result of decisions taken in first and second development stages. This is illustrated by the yellow box in Figure 26.

All the steps are same as described in the previous section of SMT 2.0, therefore only the main steps are mentioned. The differences related to the adaptation of SMT 2.0 are discussed in Chapter 4. In this section, an application of SMT 2.0 to model the behavior of communication scenario 3 for the RTOS kernel with 4 versions is presented. The scenario was applied to the 1st version among the 4 versions.

5.3.1 Generation of traces

```
trace
0,0,0,0,initKernel,1,0,0,1
0,0,0,0,createMailbox_m_5,102bd8,0,1,0
11,0,1,0,recWait,1,0,1,-1
9,0,1,-1,sendWait_s_7,1,0,1,0
9,0,1,0,sendWait_s_7,1,0,0,1
11,0,0,1,recWait,1,0,1,0
```

Figure 27: The trace logged for the communication scenario 3 of the RTOS kernel after instrumentation by SMT 2.0.

The traces were generated through instrumentation of the RTOS kernel using wrapper functions and scenarios. The traces for the communication scenario 3 of the RTOS kernel described in Chapter 4 are shown in Figure 27.
5.3.2 Creating a initial graph

The initial graph produced as an output of this step is shown in Figure 28.

5.3.3 Merging the initial graph

The merged graph shown in Figure 29 is the result of custom merge algorithm described in Chapter 4.
5.3 ADAPTING SMT 2.0 TO THE RTOS KERNEL

5.3.4 Displaying the final behavioral model

Figure 29: The merged graph for the communication scenario 3 of the RTOS kernel is obtained using the `start_merge()` function of SMT 2.0.

Figure 30: The final behavioral model for the communication scenario 3 of the RTOS kernel obtained from `make_final_efsm()` and displayed by the `draw()` function.

The final behavioural model produced is an EF SM containing data associated with the transitions as shown in Figure 30.
5.4 Result for Sub-goal 1

The first sub-goal was to develop a method to visually compare the obtained models to highlight changes in the form of different transitions.

Figure 31: This figure shows the behavioral models obtained for time scenario 2 of the RTOS kernel.

Figure 31 shows the behavioral models obtained for time scenario 2. These behavioral models are generated from version 2 and version 3 of the RTOS kernel with 4 versions. Spotting changes between those models by observation alone is difficult, as there are many transitions. Thus to justify the changes, visual comparison algorithm can be used.

Figure 32 shows the model after visual comparison. It can be seen that transitions that are different between the two models are highlighted in colors. Thus, enabling effective spotting of changes. Hence, achieving the first sub goal to develop a method to visually compare
5.5 Interpretation of Results in the Following Sections

The results in the following sections are for two different sets of RTOS kernels. The first set contains 4 versions and the second set contains 9 versions. Every result has a table and a set of figures with source code diff.

Data in the table is co-related with the figures and source code diff, and the results are presented accordingly.
5.6 RESULTS FOR SUB-GOAL 2

The second sub-goal was to analyze the behavioral changes by visually comparing the behavioral models and taking difference of the identified functionalities from source code.

5.6.1 Results from RTOS kernel with 4 versions

<table>
<thead>
<tr>
<th>Method &amp; Model</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_task()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model: Task sc 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: This Table relates the functional changes of create_task() to the change in behavioral models for task scenario 2 from version 2 to version 3, for the RTOS kernel with 4 versions.

Figure 33, shows the models obtained for task scenario 2. These models show changes in behavior from version 2 to version 3, as depicted with a color change in Table 8. This change was confirmed in the models by visual comparison. The reason for change in behavior was that, the create_task() method changed functionally from version 2 to version 3, as depicted with a color change in Table 8. The diff of create_task() is shown in Figure 33. Since, task scenario 2 was designed for this method, the functional change reflected in the behavioral model.
Figure 33: The behavioral models obtained for task scenario 2 in this figure change from version 2 to version 3 of the RTOS kernel.
5.6.2 Results from RTOS kernel with 9 versions

<table>
<thead>
<tr>
<th>Method &amp; Model</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
<th>V8</th>
<th>V9</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_wait()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model: Comm sc 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: This Table relates the functional changes of send_wait() method to the change in behavioral models for communication scenario 5 from version 5 to version 6, for the RTOS kernel with 9 versions.

Figure 34 shows the models obtained for communication scenario 5. These models change in behavior from version 5 to version 6, as depicted with a color change in Table 9. These changes were confirmed in the models by visual comparison. The reason for change in behavior was that, the send_wait() method changed functionally from version 5 to version 6, as depicted with a color change in Table 9. The diff of send_wait() is shown in Figure 34. Since, communication scenarios 5 was designed for this method, the functional changes reflected in the behavioral models.

Thus the obtained results confirm the achievement of sub goal 2, of analyzing the behavioral changes by visually comparing the behavioral models. The difference of the functionally changed methods was used to support the result.

An effective way to identify functionally changed methods is discussed in Chapter 6.
Figure 34: The behavioral models obtained for communication scenario 5 in this figure change from version 5 to version 6 of the RTOS kernel with 9 versions.
5.7 RESULTS FOR SUB GOAL 3

The third sub-goal was to investigate a mechanism for analyzing indirect behavioral changes, e.g. consider a function which uses a method X to accomplish certain functionality. How does this method influence the behaviour of embedded software, by observing the function that calls it. This is referred to analyzing indirect behavioral change.

5.7.1 Result from Kernel with 4 versions

<table>
<thead>
<tr>
<th>Method &amp; Model</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract_first()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model: time sc 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10: This Table relates the functional changes of extract_first() and wait() method to the change in behavioral models for timing scenario 2 for the RTOS kernel with 4 versions

Figure 35, shows the models obtained for timing scenario 2. These models change in behavior from version 1 to version 2, as depicted with a color change in Table 10. These changes were confirmed in the models by visual comparison. The reason for change in behavior was that, the wait() method changed functionally from version 1 to version 2, as depicted with a color change in Table 10.

From the diff of wait() method, shown in Figure 35 it can be seen that in version 2, extract_first() method was used instead of extract_obj(). Since, time scenario 2 was designed for the wait() method, the functional change is reflected in the behavioral model.

Figure 36, shows that behavioral model for timing scenario 2 also changed from version 2 to version 3. But, this time the function responsible was extract_first() as is evident from the diff in Figure 36.

The diff of extract_first() was taken after the model was obtained. Since it is a kernel space function, it is not instrumented. And thus required to be inspected as it is a part of the wait() method. So a combination of visual comparison along with difference of the non-instrumented methods was used in this context.

Thus, the extract_first() method indirectly caused behavioral changes, which were captured using combination of visual comparison of the models and difference of source code.
5.7 Results for Sub Goal 3

(a) Behavioral model of time scenario 2 for version 1

(b) Behavioral model of time scenario 2 for version 2

(c) Diff of wait() method for version 1 and version 2

Figure 35: The behavioral models obtained for time scenario 2 in this figure change from version 1 to version 2 of the RTOS kernel with 4 versions.
(a) Behavioral model of time scenario 2 for version 2

(b) Behavioral model of time scenario 2 for version 3 and 4

(c) Diff of extract_first() for version 2 and version 3

Figure 36: The behavioral models obtained for time scenario 2 change from version 2 to version 3 of the RTOS kernel with 4 versions.
5.7.2 Result from Kernel with 9 versions

<table>
<thead>
<tr>
<th>Method &amp; Model</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
<th>V8</th>
<th>V9</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerInt()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model: Time sc 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11: This table captures the functional changes undergone by TimerInt() and add() method from version 4 to version 5, and the scenarios that had change in behavioral models.

![Behavioral model](image1.png)

(a) Behavioral model of time scenario 3 for version 2 to version 4

![Behavioral model](image2.png)

(b) Behavioral model of time scenario 3 for version 5 to version 9

Figure 37: The behavioral models obtained for time scenario 3 in this figure change from version 4 to version 5 of the RTOS kernel with 9 versions.

TimerInt() method is not available for the user to call as mentioned by the RTOS kernel specifications in Chapter 9. But, is still indirectly a part of time scenario 3.

Behavioral changes in models for time scenario 3 as shown in Figure 37, were observed to have taken place from version 4 to 5, as depicted with a color change in Table 11.

The link between TimerInt() method and the change in behavior captured by the models was such that, TimerInt() method used the add() method. The add() method underwent functional changes
Figure 38: This figure shows the functional change related to deadline functionality of add() method from version 4 to version 5 for the RTOS kernel with 9 versions from version 4 to version 5, which can be seen in Figure 38. This change was particular to the functionality related to deadline.

Time scenario 3 is designed to find behavior related to deadline. It is composed of user space functions which were instrumented. Models obtained for version 1 to version 4 were different from version 5 to version 9. The difference was noted from visual comparison as shown in Figure 37.

No user space functions which were instrumented had undergone functional changes. Source code difference between version 4 and 5 was done for kernel space functions, since they were not instrumented.

It was found that add() method underwent functional changes from version 4 to version 5 as seen in Figure 38. Add method was called by Timer_int() method. Timer_int() method which gets invoked as a part of the scenario in kernel space by a ISR at every tick. Thus, indirectly influencing the behavior captured by the models.

Thus, this resulted in analyzing the indirect behavioral changes caused by add() method used within Timer_Int() method on the behaviour of embedded software (RTOS kernel) by using behavioral models and source code analysis.

Thus, from the results obtained, it confirms the achievement of sub goal 3, which was to investigate a mechanism to identify indirect changes. Visual comparison and source code difference being the mechanism, as a result of the investigation.
DISCUSSION

This chapter discusses the possibilities to extend this work and suggest changes related to this work.

6.1 VERIFICATION OF BEHAVIORAL MODEL FROM A KNOWN MODEL

If a state machine of the software being developed is available then a rule deck can be extracted from that state machine. The behavioral model of the same software can then be verified against the extracted rule deck. This method does not involve properties and thus is not comparable to model checking, which involves more advanced techniques of verification [20].

Listing 4: Assumed Rule Deck

```python
rule_deck = {
    'initKernel': ['Init'],
    'createTask': ['created'],
    'run': ['Running'],
    'task': ['Running'],
    'wait': ['wait', 'Running'],
    'LoadContext': ['Terminated', 'wait', 'Running'],
    'terminate': ['Terminated']
}
```

A possible illustration of a rule deck is shown in listing 4. This rule deck was coded as a dictionary data-structure in python, which uses key:value pairs. The key refers to the transition and the value refers to a list containing acceptable or allowed states after the transition. This rule deck was used to verify the behavioral models for timing scenarios of version 2 and version 3 and the output is shown in Figure 39.

6.2 MODELLING NON-FUNCTIONAL CHARACTERISTICS

SMT 2.0 can also be used to model behavior of non-functional features such as memory and time. To capture data related to allocation and release of memory, heap monitoring tools need to be integrated along with SMT 2.0 [21, 22]. Timing information can also be inte-
grated in the model for real time applications. The resulting model would be similar to timed automata [11].

6.3 Precise Identification of Methods That Have Functionally Changed

Differential symbolic execution utilizes symbolic execution technique for detecting behavioral changes in software by analyzing its versions. This technique precisely targets methods that have undergone changes, and leaves out the ones that have remained the same. Changes in this context are functional in nature. If a method undergoes syntactical changes, for e.g., a change in variable name, then such a method is considered to be same as previous version, and thus is neglected [6].

Differential Symbolic Execution is thus a crucial technique for automatically identifying methods that have undergone functional changes. This technique combined with instrumentation of every method in the software, would lead to detailed models with sub states giving a deeper view of the software. This can be of immense value to achieve the higher order goal of comparing behavioral changes of a software.

6.4 Randomization

Executing tailor-made scenarios has its advantages as mentioned in Chapter 4. But when it comes to exploration, random execution of functions is better suited for this purpose.

The order in which the functions will be executed is random thus may lead to sequences that are different from tailor made scenarios. Thus enabling to capture unknown behaviour. Coupled up with randomization of data, this technique will explore the behaviour of software in depth due to increased code coverage [23].

The method to perform randomization is similar to the one used in SMT 1.0, except data was not randomized, and is described in Chapter 4. But there are certain concerns which were noted due to the use of this technique for the purpose of comparing behavioral changes between versions, which are as follows.

1. Since the goal is to visually compare learned behavioural models, how would the methods that have undergone functional change be included in the sequence generated by random sequence generator.

2. While executing, what if the sequence in which the methods are executing leads to a crash.

A randomization engine can be built with those two concerns as specifications. The problem related to inclusion of functionally
changed methods in the random sequence can be ensured if the differential symbolic execution technique is adopted. The problem of program crashing can be of advantage, as it will give information about the problems regarding pre-conditions to be met for certain functions. Also a rule deck such as the one showed in Listing 4 could help to filter allowed sequences, thus overcoming the problem of crashing due to unmet dependencies among methods.

6.5 Enhancing SMT 2.0 Using Machine Learning

By using machine learning, the clustering technique can be used to cluster traces into states. Merging those states will lead to a learned behavioral model, by using the methods of SMT 2.0. Thus replacing the abstraction engine with a clustering engine to predict the states. This would open the tool to any software and even embedded software.

Thus overcoming the drawback of the abstraction engine which was a hard coded technique, and inflexible when it came to learning a model of a different software other than the intended one. Traces would be the inputs to the clustering algorithm.

6.5.1 Approach to utilize the clustering technique for SMT 2.0

In this approach each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy\[24\]. In order to decide which clusters should be joined, it is important to know how dissimilar the data points are.

This dissimilarity is decided by a distance metric, which measures the distance between two data points, and tells the measure of dissimilarity, when performing clustering.

Calculation of distances would be based on matching the trace data with the previous trace data considered. If the traces match completely, distance shall be zero. If any one element differs, then the distance is 1. Thus forming clusters which can then be numbered arbitrarily to denote states.

Integration of domain knowledge using decision tables and purity indexes \[25, 26\], may help this approach to better cluster the traces.
Figure 39: This figure shows the result after verifying the behavioral models for timing scenario of version 2 and 3 against the assumed rule deck.
CONCLUSIONS AND FUTURE WORK

The decision to use traces, scenarios, abstraction and designing a custom merge algorithm made it possible to develop SMT 2.0 which could be adapted to the RTOS kernel.

The overarching goal was to compare behavioral models. Comparison a broad topic, and behavioral models can be compared in many ways. There are numerous techniques of model or code comparison such as static analysis [27], symbolic execution [6], comparison based on accuracy [16], comparison based on quality of test suites generated [10] and so on.

Visual comparison is a specific comparison approach, thus partially full-filling the higher order goal.

The decision to include a visual comparison algorithm enabled SMT 2.0 to visually compare two behavioral models by highlighting transitions that are different between the two models.

The decision to consider a source code difference, and include functionally changed methods in the scenarios, coupled up with visual comparison algorithm, made it possible to analyze the behavioral changes by visually comparing the behavioral models.

The indirect behavioral changes were analyzed by using a mechanism that involved combining visual comparison of behavioral models and difference of the source code for non instrumented methods.

Thus, we conclude by answering the research question affirmatively, i.e., embedded software consisting of different versions can be compared for behavioral changes by comparing their behavioural models obtained from execution traces.

Since SMT 2.0 is still in its second generation, which is extremely nascent, there are shortcomings which need to addressed, which is a part of future work.

7.1 FUTURE WORK

Figure 40, shows a blueprint to proceed further with this study. If comparison of learned models has to be done effectively, the choice would be either to go ahead with symbolic execution, or to continue with traces.
If traces are used, then SMT 2.0 would need the clustering engine so as to learn behavioral models of any software. Since, Differential symbolic execution [6] is used to identify functionally changed methods, it would be logical to use FSMGen which is another tool that learns behavioral models of software using symbolic execution and predicate analysis [5], provided randomization is done to explore the software.

The learned behavioral model can then be compared using not only visual but many other techniques. Some of them being classification of correct or incorrect behavior, quality of test suit derived from a learned model and so on [16, 10].


[31] nxpd. URL: https://github.com/chebee7i/nxpd (visited on 10/14/2016).
8.1 bounded queue API

The functions of the Bounded Queue are as follows

- \textit{Queue( max size )}: The bounded queue takes a input to determine its maximum size.
- \textit{enqueue( n )}: This function accept a input n and adds it to the queue.
- \textit{dequeue()}: This function removes a data element from the queue.
- \textit{isEmpty()}: This function returns true if the queue size was zero and false otherwise.
- \textit{isFull()}: This function returns true if the queue was size equal to max size and false otherwise.
- \textit{size()}: This function returns the size of the queue.

8.2 function set

Functions in the set of bounded queue API were as follows:
fl = [ enqueue, isEmpty, isNotEmpty, isFull, dequeue, size ].

8.3 indices

The indices of the functions were as follows:

- 0: enqueue()
- 1: isEmpty()
- 2: isNotEmpty()
- 3: isFull()
- 4: dequeue()
- 5: size()

Thus for a random sequence [2, 3, 1, 5], the methods isNotEmpty(), isFull(), isEmpty(), size() get executed.
APPENDIX B

This Appendix is about the specification of the RTOS kernel API functions.

The functions are divided into 3 groups, viz. Task Administration, Inter-process communication and Timing functions.

9.1 TASK ADMINISTRATION

exception init_kernel()

This function initializes the kernel and its data structures and leaves the kernel in start-up mode. The init_kernel call must be made before any other call is made to the kernel.

Argument
none

Return parameter
Int: Description of the function’s status, i.e. FAIL/OK.

exception create_task(void(*task_body)(), uint deadline)

This function creates a task. If the call is made in startup mode, i.e., the kernel is not running, only the necessary data structures will be created. However, if the call is made in running mode, it will lead to a rescheduling and possibly a context switch.

Argument
*task_body: A pointer to the C function holding the code of the task.
deadline: The kernel will try to schedule the task so it will meet this deadline.

Return parameter
Int: Description of the function’s status, i.e. FAIL/OK.
void run()

This function starts the kernel and thus the system of created tasks. Since, the call will start the kernel it will leave control to the task with tightest deadline. Therefore, it must be placed last in the application initialization code. After this call the system will be in running mode.

**Argument**
none

**Return parameter**
none

void terminate()

This call will terminate the running task. All data structures for the task will be removed. Thereafter, another task will be scheduled for execution.

**Argument**
none

**Return parameter**
none

9.2 INTER-PROCESS COMMUNICATION

Mailbox* create_mailbox(int nof_msg, int size_of_msg)

This call will create a Mailbox. The Mailbox is a FIFO communication structure used for asynchronous and synchronous communication between tasks.

**Argument**
nof_msg: Maximum number of messages the mailbox can hold.
Size_of_msg: The size of one Message in the Mailbox.

**Return parameter**
Mailbox*: a pointer to the created mailbox or NULL.
exception remove_mailbox(Mailbox* mBox)

This call will remove the mailbox if it is empty and return OK. Otherwise no action is taken and the call will return NOT_EMPTY.

**Argument**
Mailbox*: A pointer to the Mailbox to be removed.

**Return parameter**
OK: The mailbox was removed
NOT_EMPTY: The mailbox was not removed because it was not empty.

exception send_wait( Mailbox* mBox, void* Data)

This call will send a message to the specified mailbox. If there is a receiving task waiting for a message on the specified mailbox, send_wait will deliver it and the receiving task will be moved to the Readylist. Otherwise, if there is no receiving task waiting for a message on the specified mailbox, the sending task will be blocked. In both cases (blocked or not blocked), a new task schedule is done and possibly a context switch. During the blocking period of the task its deadline might be reached. At that point in time the blocked task will be resumed with the exception: DEADLINE_REACHED. Note: send_wait and send_no_wait messages shall not be mixed in the same mailbox.

**Argument**
*mBox a pointer to the specified mailbox.
*Data: a pointer to a memory area where the data of the communicated message is residing.

**Return parameter**
exception: The exception return parameter can have two possible values:
- OK: Normal behavior, no exception occurred.
- DEADLINE_REACHED: This return parameter is given if the sending task’s deadline is reached while it is blocked by the send_wait call.
exception receive_wait( Mailbox* mBox, void* Data)

This call will attempt to receive a message from the specified mailbox. If there is a send_wait or a send_no_wait message waiting for a receive_wait or a receive_no_wait message on the specified mailbox, receive_wait will collect it. If the message was of send_wait type the sending task will be moved to the Readylist. Otherwise, if there is no send message (of either type) waiting on the specified mailbox, the receiving task will be blocked. In both cases (blocked or not blocked), a new task schedule is done and possibly a context switch. During the blocking period of the task its deadline might be reached. At that point in time the blocked task will be resumed with the exception: DEADLINE_REACHED.

Argument
*mBox: a pointer to the specified mailbox.
*Data: a pointer to a memory area where the data of the communicated message is to be stored.

Return parameter
exception: The exception return parameter can have two possible values:
- OK: Normal function, no exception occurred.
- DEADLINE_REACHED: This return parameter is given if the receiving task’s deadline is reached while it is blocked by the receive_wait call.

exception send_no_wait( Mailbox* mBox, void* Data)

This call will send a message to the specified mailbox. The sending task will continue execution after the call. When the mailbox is full, the oldest message will be overwritten. The send_no_wait call will imply a new scheduling and possibly a context switch. Note: send_wait and send_no_wait messages shall not be mixed in the same Mailbox.

Argument
*mBox: a pointer to the specified mailbox.
*Data: a pointer to a memory area where the data of the communicated message is residing.

Return parameter
Description of the function’s status, i.e. FAIL/OK.
exception receive_no_wait(Mailbox* mBox, void* Data)

This call will attempt to receive a message from the specified mailbox. The calling task will continue execution after the call. When there is no send message available, or if the mailbox is empty, the function will return FAIL. Otherwise, OK is returned. The call might imply a new scheduling and possibly a context switch.

**Argument**
- *mBox*: a pointer to the specified mailbox.
- *Data*: a pointer to the message.

**Return parameter**
Integer indicating whether or not a message was received (OK/FAIL).

### 9.3 Timing Functions

exception wait(uint nTicks)

This call will block the calling task until the given number of ticks has expired.

**Argument**
- nTicks: the duration given in number of ticks

**Return parameter**
- exception: The exception return parameter can have two possible values:
  - OK: Normal function, no exception occurred.
  - DEADLINE_REACHED: This return parameter is given if the receiving tasks’ deadline is reached while it is blocked by the receive_wait call.
void set_ticks( uint nTicks)

   This call will set the tick counter to the given value.

   Argument
   nTicks: the new value of the tick counter

   Return parameter
   none

uint ticks(void)

   This call will return the current value of the tick counter

   Argument
   none

   Return parameter
   A 32 bit value of the tick counter

uint deadline(void)

   This call will return the deadline of the specified task

   Argument
   none

   Return parameter
   the deadline of the given task

void set_deadline(uint deadline)

   This call will set the deadline for the calling task. The task will be
   rescheduled and a context switch might occur.

   Argument
   deadline: the new deadline given in number of ticks.

   Return parameter
   none
void TimerInt(void)

This function is not available for the user to call. It is called by an ISR (Interrupt Service Routine) invoked every tick. Note, context is automatically saved prior to call and automatically loaded on function exit.

void SaveContext (void)

This function is hardware dependent. All relevant registers are saved to the TCB of the currently running task.

void LoadContext (void)

This function is hardware dependent. All relevant registers are restored from the TCB of the currently running task to the CPU registers.

void timer_isr (void)

This function is not available for the user to call. It is an ISR (Interrupt Service Routine) invoked every tick. Note. It calls the C-Function TimerInt().
This Appendix is about the SMT 2.0 tool.

SMT 2.0 was programmed in python using Pycharm 2016 by jetbrains [28].

Additional libraries used by the tool are mentioned below.

1. NetworkX: This library has python data structures for graphs, digraphs and multi-digraphs [29].
2. GraphViz: This is a visualization library [30].
3. nxpd: This library is used for visualizing NetworkX and GraphViz graphs [31].

The Architecture of SMT 2.0 is as follows.

1. Trace reader: This block reads the traces from the text file and generates the complex_state_list
2. Abstraction Engine: This block analyzes the raw data in complex_state_list and converts it into states which are a part of simple_state_list.
3. XDFA: This block has a class automaton which contains all the functions related to seeking information of the graphs, such as
getting the number of total nodes in the graph, the neighbours of a node etc.

4. Create Initial Graph: This block is responsible for creating the initial graph from the simple_state_list.

5. Merging and Filtering: This block merges the initial graph into a state machine.

6. Model Display: This block displays the state machine.

7. Compare and Visualize: This block is responsible for comparing two state machines and visualizing them.
DECLARATION

I hereby declare that the work presented in this document is true to the best of my knowledge.

*Halmstad, Sweden, September 15, 2016*

___________________________________________________________________________

Satej Vinayak Khandeparkar,
July 5, 2017