Verification of UML State Diagrams using a Model Checker

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Yiwei Zou

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

August, 2013
Verification of UML State Diagrams using a Model Checker

By Yiwei Zou

We recommend acceptance of this manuscript in partial fulfillment of this candidates requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Dr.Kasi Periyasamy
Examination Committee Chairperson

Dr.Mao Zheng
Examination Committee Member

Dr.Tom Gendreau
Examination Committee Member
Abstract


This manuscript describes the process of verifying the UML state diagrams by using model checker. A UML state diagram describes the behavior of an object which includes a sequence of states that the object visits during its lifetime. Verification of UML state diagram is important because if one state diagram is incorrect, the object’s behavior will not be displayed correctly which leads to incorrect coding and eventually may lead to the possible failure of the system. However, it is hard to verify UML state diagram without the aid of other tools. Therefore, model checker is introduced to verify the UML state diagram. Since model checkers use different syntax, one needs to convert state diagrams from UML tools to the syntax used by the model checker. Moreover, model checkers also use temporal logic expressions for verification which need to be added to the information in the state diagram for verification purposes. This manuscript describes the design and implementation of a conversion tool that converts UML state diagrams drawn using Visual Paradigm to the syntax used by UPPAAL model checker. The tool was tested with several UML state diagrams.
Acknowledgements

I would like to express my sincere thanks to my project advisor Dr. Kasi Periyasamy for initiating this project and providing his valuable guidance. I would also like to express my appreciation to the Department of Computer Science at the University of Wisconsin-La Crosse for providing the learning materials and computing environment for my project.
## Contents

1 Introduction .................................................. 1
   1.1 UML State Diagram ..................................... 1
   1.2 Model Checker .......................................... 3

2 Current Project ............................................. 5
   2.1 Overview ................................................ 5
   2.2 UML Tool ............................................... 5
   2.3 Model Checking Tool ................................... 6

3 Converting Tool ............................................. 7
   3.1 Development Approach ................................. 7
   3.2 Convert Function ....................................... 9
   3.3 User Interface ......................................... 12
      3.3.1 Main Window ...................................... 12
      3.3.2 Additional Information Window .................. 15

4 Verification ................................................ 17
   4.1 Simulator ............................................... 17
   4.2 Verifier ............................................... 20
   4.3 Verify Command ....................................... 22
      4.3.1 Reachability Properties ......................... 22
      4.3.2 Safety Properties ................................. 23
      4.3.3 Liveness Properties ............................... 24
      4.3.4 Deadlock Properties .............................. 26

5 Case Study .................................................. 26
   5.1 Coffee Machine ....................................... 26
      5.1.1 Convert ............................................ 26
      5.1.2 Verify ............................................. 28

6 Future Work ................................................ 30

7 Conclusion ................................................ 32

8 References .................................................. 33

9 Appendix .................................................... 34
   9.1 CD Player .............................................. 34
   9.2 Coffee Machine ....................................... 35
   9.3 Cruise Controller ...................................... 36
   9.4 ATM Machine .......................................... 37
   9.5 Traffic Light .......................................... 38
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A template of a UML state diagram</td>
</tr>
<tr>
<td>2</td>
<td>A finite-state system</td>
</tr>
<tr>
<td>3</td>
<td>The simulator on a finite-state system</td>
</tr>
<tr>
<td>4</td>
<td>The temporal logical formula to verify the finite-state system</td>
</tr>
<tr>
<td>5</td>
<td>Overview of the project</td>
</tr>
<tr>
<td>6</td>
<td>UPPAAL Sample</td>
</tr>
<tr>
<td>7</td>
<td>Visual Paradigm XML file</td>
</tr>
<tr>
<td>8</td>
<td>UPPAAL XML file</td>
</tr>
<tr>
<td>9</td>
<td>Main Window</td>
</tr>
<tr>
<td>10</td>
<td>Additional Information Window</td>
</tr>
<tr>
<td>11</td>
<td>Simulator Example</td>
</tr>
<tr>
<td>12</td>
<td>Simulator Trace Example</td>
</tr>
<tr>
<td>13</td>
<td>Simulator Trace Example</td>
</tr>
<tr>
<td>14</td>
<td>Exist Eventually</td>
</tr>
<tr>
<td>15</td>
<td>Exist Globally</td>
</tr>
<tr>
<td>16</td>
<td>Always Globally</td>
</tr>
<tr>
<td>17</td>
<td>Always Eventually</td>
</tr>
<tr>
<td>18</td>
<td>Always Leads To</td>
</tr>
<tr>
<td>19</td>
<td>Coffee Machine State Machine Diagram</td>
</tr>
<tr>
<td>20</td>
<td>Coffee Machine Convert</td>
</tr>
<tr>
<td>21</td>
<td>Coffee Machine In UPPAAL</td>
</tr>
<tr>
<td>22</td>
<td>Coffee Machine Verify Queries</td>
</tr>
<tr>
<td>23</td>
<td>CD Player state diagram</td>
</tr>
<tr>
<td>24</td>
<td>CD Player finite-state system</td>
</tr>
<tr>
<td>25</td>
<td>Coffee Machine state diagram</td>
</tr>
<tr>
<td>26</td>
<td>Coffee Machine finite-state system</td>
</tr>
<tr>
<td>27</td>
<td>Cruise Controller state diagram</td>
</tr>
<tr>
<td>28</td>
<td>Cruise Controller finite-state system</td>
</tr>
<tr>
<td>29</td>
<td>ATM Machine state diagram</td>
</tr>
<tr>
<td>30</td>
<td>ATM Machine finite-state system</td>
</tr>
<tr>
<td>31</td>
<td>Traffic Light state diagram</td>
</tr>
<tr>
<td>32</td>
<td>Traffic Light Left Lane state diagram</td>
</tr>
<tr>
<td>33</td>
<td>Traffic Light finite-state system</td>
</tr>
<tr>
<td>34</td>
<td>Traffic Light Left Lane finite-state system</td>
</tr>
</tbody>
</table>
GLOSSARY

UML

Unified Modeling Language - A standardized, general-purpose modeling language in the field of software engineering.

Temporal Logic

Any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time.

DOM

Document Object Model - A cross-platform and language-independent convention for representing and interacting with objects in HTML, XML and XHTML documents.

Finite-state System

A mathematical model of computation used to design both computer programs and sequential logic circuits.
1 Introduction

1.1 UML State Diagram

The UML state diagram displays the behavior of an object which specifies the sequence of states the object visits during its lifetime in response to events, together with its responses to those events. [1]

The elements a UML state diagram contains are listed below:

• State: A state is a condition during the life of an object. When an object is in a particular state, it satisfies some condition, performs some activity, or waits for some external event.

• Transition: A transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state. Mostly, a transition has a source state, an event trigger, an action and a target state as components.

A UML state diagram is a mixture of graphical and textual representation that precisely captures both the state topology and its actions. It is strongly connected to other UML diagrams since it gives users a more specific view of the behavior of an instance of one particular class in the class diagram. However, a UML state diagram is not completely descriptive. Any complex diagram requires a large amount of textual information like the specification of actions and guards. Moreover, the UML state diagram is meant for design purposes and the semantics of application is not easy to observe from the diagram.

A sample UML state diagram is shown in figure 1. The diagram displays the behavior of a CD player. There is one vital error in the diagram. There is no transaction from the
state “CDStopped” to the state “CDPlaying” which indicates that there is no way for the CD player to move from stop to play. However, it takes time for users to verify the diagram just by looking at it. Moreover, there could still be hidden error even if the user checks the diagram carefully. This diagram includes only five simple states. The problem would be much more complex if the number of states increases or complex state diagrams are used.

Embedding verification techniques in UML may solve the problems. However, UML is meant as a design notation that already has a vast number of design notations. Moreover, design and verification are usually performed separately. Therefore, it is preferable to take UML design diagram as an input to some verifier and then verify the design.
1.2 Model Checker

In computer science, model checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. Typically, the specification contains several requirements such as the absence of deadlocks and critical states that could cause the system to crash. Model checking, therefore, is a technique that can automatically verify correctness properties of finite-state systems. [2]

Model checking technology arguably ranges among the foremost applications of logic to computer science and computer engineering. In the twenty-five years since its invention, model checking has achieved multiple breakthroughs, bridging the gap between theoretical computer science, hardware and software engineering. Today, model checking is extensively used in the hardware industry, and has become feasible for verifying many types of software as well. Model checking has been introduced into computer science curricula at universities worldwide, and virtually has become a universal tool for the analysis of systems. [6]

The finite-state system of the CD player in the previous section is shown in figure 2. The finite-state system is similar to the UML state diagram. However, it has some additional information for the guards and synchronizations to actually run the finite-state system.

Model checkers have two ways to verify the finite-state system. One way is to use a simulator to track the behavior of the object, as shown in figure 3. The other is using a verifier. By typing in the specification as a temporal logical formula, the verifier will exhaustively go into all reachable states to verify whether the system meets the specification. The example temporal logical formula is shown in figure 4.
Both the simulator and verifier will be discussed later in this manuscript.

Figure 2: A finite-state system

Figure 3: The simulator on a finite-state system
2 Current Project

2.1 Overview

The UML state diagram is a great way to view the behavior of an object in a complex system. However, it is difficult to verify whether or not the UML state diagram is correct. The Model checkers are developed to verify if a finite-state system meets given specifications. So, the main purpose of this project is to develop a tool to convert a UML state diagram into a finite-state system that can be used as input to the model checking tools so that the UML state diagram could be verified in the model checker.

Therefore, this project could be divided into two parts:

- Develop a tool to convert UML state diagram into finite-state system that model checkers will accept.
- Verify the converted UML state diagram using the model checker.

2.2 UML Tool

The first step of the project is to convert UML state diagrams into the syntax accepted by the model checker. For this project, the developer chose Visual Paradigm as the UML tool. There is no definite advantage of choosing Visual Paradigm for this project. In fact,
the method will work for any UML tool. The state diagram for the application is drawn using the Visual Paradigm tool and saved as an XML file. This file contains the graphical information as well as the state machine information.

2.3 Model Checking Tool

There are a number of model checking tools available today. For this project, the UPPAAL model checker has been selected. There is no definite reason for choosing UPPAAL. It is a free tool and serves the purpose for this project.

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types like integers, arrays, etc. UPPAAL has an easy-to-use GUI for simulation and verification of finite-state systems.

UPPAAL is developed jointly by Basic Research in Computer Science at Aalborg University in Denmark and the Department of Information Technology at Uppsala University in Sweden. Since its first release in 1995, UPPAAL has been applied in a number of case studies. And to meet the requirement arising from the case studies, the tool has been extended with various features. [5] A sample state diagram drawn using UPPAAL is shown.
in figure 6.

Figure 6: UPPAAL Sample

3 Converting Tool

3.1 Development Approach

An agile software development methodology was used in developing the converting tool. The agile methodology requires strong developer-customer communication. The requirements for this part of the project can be divided into these parts:

- Extract the necessary information from the UML state diagram’s XML file that is created by the Visual Paradigm.
• Use the information that is extracted to create an XML file that can be read by UPPAAL to create an equivalent finite-state system for the user to verify.

• Create the GUI that supports the transition of state diagram from Visual Paradigm to UPPAAL and also facilitate verification.

Though the requirements are clear at the beginning, the model checking methodology is a whole new thing for the developer and extracting information and creating new diagram require the developer to learn while developing. Therefore, agile methodology was chosen for this part of the project.

The advisor, Dr. Kasi Periyasamy, acted as the customer representative and provided comments and feedback throughout the development of the tool. The developer created a simple system first. Improvements and other features were all added after the developer had a discussion with the advisor so that the tool would be developed based on a common understanding between the customer and the developer.

The basic purpose for this part of the project was to convert the UML state diagram into the finite-state system. As an agile method was used, the entire set of functionalities was defined and developed throughout the development process. Feedback from earlier stage was incorporated into future stages.

The functionalities of the tool are as listed:

• Get the import file

• Get the picture of the import UML state diagram

• Open Visual Paradigm and UPPAAL directly from the tool

• Create guidance for user to add additional information that is needed for converting
• Convert the import file to export file

• Display the export file and save the export file in given location

3.2 Convert Function

The most significant part of this part of the project is to develop the convert function from UML State Diagram to a finite-state system accepted by UPPAAL.

Therefore, DOM is introduced to manipulate the XML files.

DOM, short for the Document Object Model, which is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents. Usually, DOM is used to deal with web files. However, DOM can be used here to extract useful information from the xml file that is created by the UML drawing tool, the Visual Paradigm.

DOM defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term “document” is used in the broad sense, increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.

As in figure 7, in Visual Paradigm XML file, there are several of lines of code, much of it is not relevant for the current project. The most important information needed for the conversion process are visual information about each symbol on the diagram, and the semantic information such as state, transition, and guard. Therefore, what is needed to do is
just to find out information required and get them out of the raw file. By using DOM, steps are simple. First step is to find the location of the information needed, like it is under which node, what is the parent node of this node, etc. Then, use DOM to get these information and store them in arraylists for further usage.

After these information are extracted, the other end of this function is to create a whole new XML file which could be read by UPPAAL by using these information and other additional information that the user needs to add, like the value of variables, and the implementation of the actions in UML state diagram.

From the image in figure 8, the information needed to create a UPPAAL XML file are listed as follows:
• Each state’s name and invariant

• Each transition’s name

• Each transition’s guard

• Each transition’s activity

• The initial value of the variables

• The implementation of each activity

For the first four elements, we can extract them from the Visual Paradigm file. However, for the last two elements, additional information is needed from the user. It is not user-friendly to just simply ask the user to input these information. For things like activity implementation, guidance is needed for guiding the user to put in correct information that is required.

By combining the information from the Visual Paradigm file and the information added by user, it will be easy for us to generate the correct UPPAAL file.
3.3 User Interface

The converting tool has two windows excluding some minor interface like error message, etc. One is the main interface in which the user can input the Visual Paradigm diagram and file, open Visual Paradigm and UPPAAL, add additional information, and see the result of converting. The other is the add additional information page, in this page, guidance is provided for user to write in additional information that cannot be provided from the Visual Paradigm file.

3.3.1 Main Window

The main window consists of the basic functions that are required. The upper part displays the functions related to Visual Paradigm. The lower part displays the functions related to UPPAAL. The function of each button, label or else is listed below:

- The button “Select a file”, is used for users to select a visual paradigm file from the disks. By clicking it, a file chooser window will be opened and the user can select the file needed. The file chooser can only choose xml files.

- The blank area below the “Select a file” button will show the image of UML state diagram that is selected in the “Select a file” button.

- The directory of choosen file will be shown next to the “Select a file” button.

- The “Open VP” button will, literally, open Visual Paradigm directly from the conversion tool.
• The “Previous” and “Next” button is used to select from a list of UML state diagram if there is more than one image.

• The “Add additional Info” button is used to add additional information that is needed to create the file for finite-state system. By clicking it, an “Add additional information” window will pop up.

• The “Convert” button is used to convert the UML state diagram file into the finite-state system file. The user can still create the file without entering additional information. However, the exported file is less likely to function normally and the user will need to add additional information in UPPAAL if the user choose not to add additional information here.

• The “Convert This Diagram Only” button is used to convert only the diagram shown in the area above into finite-state system if there are more than one diagrams. If there is only one state diagram in the Visual Paradigm file, the button will function the same as “Convert” button. As discussed above, not add additional information will affect the function too.

• The “Select Save Place” button is used to select the place to save the exported file. By clicking it, a file chooser will be opened and the user can select the place to save the file.

• The text area below the “Select Save Place” button will show the exported file once “Convert” button or “Convert This Diagram Only” button is clicked.

• The “Open UPPAAL” button will open the UPPAAL in the PC if the PC has UPPAAL installed.
Figure 9: Main Window
3.3.2 Additional Information Window

The Additional Information Window is shown as in figure 10. The user can add the additional information that is needed in the converting process. The function of each part is listed below:

- In the “Local Param” area, the user can add local parameters that should be added in the converted file. The added parameters will be listed in the text area below.

- In the “Input Param” area, the user can add input parameters that should be added in the converted file. The added parameters will be listed in the text area below.

- In the “Function” area, the user can add functions that should be added in the converted file. The added functions will be listed in the text area below.

- The text area at right will display the guidance information needed for users to add information. By clicking the “Show Information” button, the guards and activities of the UML state diagram will be listed in the text area.

As might be confused, the local parameter differs from input parameters that the local parameter is the parameter that has initial value at the start of the system and this value cannot be changed by the user at the start of the system. While the input parameter is the parameter that needs the user to set initial value every time the system runs. For example, as in traffic light, the direction of the traffic light, whether it is north-south or west-east, is an input parameter. However, whether there is car in the left lane is local parameter. Since the car will keep coming in the left lane and the user cannot determine it in the beginning.

The functions are of the same meaning of updates in UPPAAL and activities in Visual Paradigm. It is usually very simple. For instance, in the traffic light example, there might
be an activity which is “UpdateLeftLane(car)”, in which the function is only used to set the left lane to empty since the green left arrow has been turned on.

Figure 10: Additional Information Window
4 Verification

The other part of the project is to verify the correctness of the converted diagram. Basically, UPPAAL has a simulator and a verifier to assist users on verifying the real-time finite-state system. The simulator is used to examine the system in an interactive and graphical manner. It allows the user to go into the system step by step. The verifier is the place where the user enters the specification and then UPPAAL will verify it by running through the system in either depth-first way or breadth-first way. Both of them can be activated by clicking on the corresponding tab in the menu.

4.1 Simulator

As is discussed before, the simulator can guide users to explore the dynamic behavior of the system in an interactive way. Different from the verifier, which examines all the path, reachable states and behaviors of the system, the simulator only creates trace, which is a sequence of states of the system, that is one particular execution path.

The simulator is useful in following ways:

- In early stage of modeling, the simulator provides an inexpensive mean for users to get familiar with the system.

- By creating particular traces, the simulator allows the user to intentionally step in the particular places that the user want to examine. Therefore, it helps in fault-detection.

- After an error detected by verifier, the user can use the simulator to visualize the wrong trace that causes the error.
• In comparison to the simulator, the verifier is obviously much more expensive since it will do an exhaustive simulation of the system. Therefore, the simulator is a light-weight tool than verifier.

![Simulator Example](image)

Figure 11: Simulator Example

As shown in figure 11, the graphical user interface of simulator is divided into four parts:

• The control part, which is the left panel. It is used to choose enabled transitions, go through a trace, and toggle the random simulations.

• The variable view, which is the middle panel. It is used to show the value of the variables like the clock constraints.

• The system view, which is the up-right panel. It shows the instantiated automata, active locations of the current state and the next state it could go to.

• The message sequence chart, which is the down-right panel. It shows the synchronisations between the different processes and the states the system goes to at every
Conclusively, since the finite-state system we are using now is directly converted from the UML state diagram, the simulator is a great way for the users to understand the system. The other great usage of the simulator is to create diagnostic traces that typically displays the error in the system. Figure 12 shows the trace of a particular path.

Figure 12: Simulator Trace Example
4.2 Verifier

The verifier in UPPAAL accepts commands that is written in the UPPAAL requirement specification language. The semantics of this simple language is in the appendix. We will discuss this in the next section. These commands represents the specification that needs to be checked in the finite-state system.

As is shown in figure 13, the graphic user interface of the verifier is divided into four parts:

- The first part is the overview, it displays all the queries that is executed or is going to be executed. As may be noticed, there is a signal light at the end of every query. If the signal is grey, it means the property has not been checked yet. If the signal is red, it means the property is not satisfied. If the signal is green, it means the property is satisfied. The last color is yellow, it means the system cannot determine the status of
the property with the approximation used. There are four buttons on the right side, by which the user can check, add or delete queries.

- The next part is the query, it displays the selected query. The user can modify the query in this place.

- The third part is the comment, it displays the comment of the selected query. The user can modify the comment in this place.

- The last part is status, it displays the system logs of every model check operations. The user can trace it back to see the model check history. And see the result of each model check operation.
4.3 Verify Command

Basically, the UPPAAL query language is simple, which leads to the result that the types of properties that can be directly checked using the UPPAAL queries are simple. The UPPAAL designers have intentionally taken this approach instead of allowing complex queries in order to improve the efficiency of the tool. For this reason, the verification of complex properties will need the checking of many simple different queries.

As is discussed before, UPPAAL query language is quite simple. The specific types of properties which can be expressed in UPPAAL query language can be classified into these four types:

- Reachability properties: A specific condition holds in some state of the model’s potential behaviors.
- Safety properties: A specific condition holds in all the states of an execution path.
- Liveness properties: A specific condition is guaranteed to hold eventually or at some moment.
- Deadlock properties: A deadlock is possible or not.

These four types will be discussed in the next four paragraphs.

4.3.1 Reachability Properties

As discussed above, reachability property is a specific condition holds in some state of the model’s potential behaviors.

In UPPAAL query language, they are expressed in the form as “E⟨⟩ p”, which is “Exists
eventually p", meaning that there is an execution path in which p eventually holds. As shown in figure 14.

An example for this is “E\(\cdot\) CDPlayer1.CD == true”, meaning that the value of CD is true in at least one state of at least one execution path.

Figure 14: Exist Eventually

4.3.2 Safety Properties

Slightly different from reachability properties, a safety property is a specific condition that holds in all the states of an execution path.

There are two kinds of safety properties:

- **E[] p**: “Exist globally p”, meaning there is a specific execution path in which p holds for all the states of the path, as shown in figure 15. For instance, “E[] CDPlayer1.CD == true” means that the value of CD is true in all the states for at least one execution path.

- **A[] p**: “Always globally p”, meaning for all the execution paths, p holds for all the states, as shown in figure 16. For instance, “A[] CDPlayer1.CD == true” means that
the value of CD is always true in all the states of all execution paths.

4.3.3 Liveness Properties

A liveness property means that a specific condition is to be hold eventually in every path.

In UPPAAL query language, there are two kinds of liveness properties:

- $A\langle\rangle p$: “Always eventually $p$”, which means for all the execution paths, $p$ holds eventually in these paths. In other word, $p$ holds for at least one state for every execution path, as shown in figure 17. For instance, “$A\langle\rangle \text{CDPlayer1.CD == true}$”
means that for all paths, the value of CD will be true for at least one state.

Figure 17: Always Eventually

q → p: “q leads to”, which is better to understand if we add always in the meaning, which is “q always leads to p”. Every path that starts with q will eventually enter states that p holds, as shown in figure 18. For instance, “CDPlayer1.CD == true → CDPlayer1.CDPlaying” means that after reaching the state that the value of CD is true, the path will always go the state CDPlaying.

Figure 18: Always Leads To
4.3.4 Deadlock Properties

As is obviously shown in the title of this subsection, UPPAAL query language provides deadlock check which checks if a deadlock is possible or not in the system.

As in its normal understanding, a state is in a deadlock if it is impossible that the model evolves to a successor state neither by wait some time nor by a transition between locations.

The user can use deadlock check in any of the above properties. For example, A[] not deadlock, meaning “There is no deadlock in the system”, is frequently used in system model checking.

5 Case Study

5.1 Coffee Machine

Coffee machine is a typical simple example. By going through the process from converting the UML state diagram to verify the finite-state system generated from the UML state diagram, the user can get a well understanding of the whole project. The UML state diagram for the coffee machine is shown as in figure 19.

5.1.1 Convert

The main focus of converting part is to add additional information since others are handled automatically by the tool.
As shown in figure 20, the guidance information provides all the guards and activities in the UML state diagram. By which, the user can find the parameters that needs to be declared such as cashDeposit, cups, CoffeeAvailable, etc. And these easy to be seen variables can all be set as input parameters since their values can be defined by user at the beginning of the system starts, like the number of cups in the machine. And there are boolean values shown in the guidance information. Consider more, these boolean values are the value of cream and sugar. Therefore, local parameters are set which are “bool cream = false” and “bool sugar = false”.

For the function, it is required that all the activites listed right should be defined here. However, there are usually many redundant activities in the UML state diagram. For this one, the function “GetCoffee(bool, bool)” is repeated four times in the guidance information. Usually, the function can be identified by the name of it. For example, the function “SetSugarSelected” will just set the boolean value of sugar to true.
5.1.2 Verify

The converted file is shown in figure 21. Compare it with the one in UML, there are some differences but mostly they are alike.

I have added two verify queries. One is “A[] cups >= 0”, which means that for all time, the value of cups are always greater than 0, which should be true because when the number of cups are less than 0, the system should go to “NotInService” and stop the service. However, as the red light indicates that this statement is wrong. By checking the diagram using simulator, the problem turns out to be that there are no guards when the channel “CoffeeServed” is called. Therefore, in finite-state system, it means that when the value
of cups is 0, the system can go to “NotInService” which is expected but also can go to “ReadyForService” since no guard is in that transition. The solution is simply add guards “cups > 0 && CoffeeAvailable > 0 && CashAvailable > 0” to the transition. And the guard should also be added back in the UML diagram.

The other is “CM1.SugarSelected → CM1.PouringCoffeeWithNoSugarNoCream”, which should be wrong. Because with sugar selected, the system cannot pour coffee with no sugar and no cream. And here it is wrong there it is proved to be true.

![Figure 21: Coffee Machine In UPPAAL](image1)

![Figure 22: Coffee Machine Verify Queries](image2)
6 Future Work

This project aims on converting the UML state diagram to finite-state system and verifying it using the model checker. For the conversion tool, it has basically finished all the functionalities that is required. However, there are still some areas that could be enhanced to make the user experience better and to make the tool more powerful. For the verification part, simulators and verifiers are both well-introduced. However, some additional semantics are not introduced in this project and it can be added in the future work. Some of these improvements and enhancements are described in this section.

For this project, the developer has tested various examples including CD player, ATM machine, traffic light, coffee machine, cruise controller, etc. Some of them are introduced in the paper while others are not. The UML diagrams and converted finite-state system is displayed in the appendix. It is expected that some more cases will be tested.

Functionalities to convert multiple state diagrams are not currently implemented. Some of the functions have been written about converting multiple diagrams. However, it does not guarantee that the converted diagram works properly. Mostly, the problem comes from the synchronization between different diagrams.

Currently, the tool uses the widgets in Netbeans to assemble the GUI. Therefore, it does not look pretty and is somehow messy. Improving the GUI could easily make the tool look better.

The tool only supports converting from Visual Paradigm to UPPAAL. However, there are some other UML drawing tools and model checkers which are popular such as Rational Rose, PAT, etc. Therefore, it will be better if we can make the tool to convert from multiple UML drawing tools to multiple model checkers.
As may be noticed, there is not error checking in adding additional information. Currently, the tool cannot check if we add the wrong information in the diagram. Users can correct it in UPPAAL after they run it. However, it could be better if we make the error checking in the converting tool.

For the UPPAAL query language, basic semantics have been introduced and used. However, as discussed above in verify part, UPPAAL supports a very simple semantic which means UPPAAL only supports model check simple specifications. If a complicated specification is required to be checked, it needs to be resolved into several small and simple specifications. And this resolve process is not introduced in the paragraph.
7 Conclusion

This paper describes the converting process from UML state diagram to finite-state system and the verifying process of the finite-state system. UML drawing tools do provide synchronization between different diagrams like class and use case diagram. However, these tools do not check whether the UML state diagram is right or not. For simple diagram, the user may check it manually. If the diagram is complicated, it is difficult for users to find inner problems in the state diagram. The problem may cause problems like deadlock when building the program using the UML state diagram. Therefore, a solution is provided in this paper that the state diagram can be converted into finite-state system semiautomatically. After that, the model checking tools can check it by going into every path automatically.

Several case studies have been described in this paper. They are great for users to learn about the verification of the model checker and the function of the converting tool that is created in this project.
8 References


9 Appendix

9.1 CD Player

Figure 23: CD Player state diagram

Figure 24: CD Player finite-state system
9.2 Coffee Machine

Figure 25: Coffee Machine state diagram

Figure 26: Coffee Machine finite-state system
9.3 Cruise Controller

Figure 27: Cruise Controller state diagram

Figure 28: Cruise Controller finite-state system
9.4 ATM Machine

Figure 29: ATM Machine state diagram

Figure 30: ATM Machine finite-state system
9.5 Traffic Light

Figure 31: Traffic Light state diagram

Figure 32: Traffic Light Left Lane state diagram
Figure 33: Traffic Light finite-state system

Figure 34: Traffic Light Left Lane finite-state system