DESIGN VALIDATION OF MULTI-MODE SYSTEMS

by

Diyang Chu

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2013
As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Diyang Chu entitled Design Validation of Multi-mode Systems and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Jonathan Sprinkle
Date: 20 November 2013

Srinivasan Ramasabramanian
Date: 20 November 2013

Ricardo Sanfelice
Date: 20 November 2013

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College. I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Jonathan Sprinkle
Date: 20 November 2013
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. This work is licensed under the Creative Commons Attribution-No Derivative Works 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

SIGNED:  Diyang Chu
ACKNOWLEDGEMENTS

First, I wish to acknowledge my research advisor Dr. Jonathan Sprinkle for his patient and unselfish help for the past five years. Without his encouragement and support, this dissertation will not be possible.

I am also grateful to all my friends that helped me maintain the balance between research and off-campus life. They make my graduate school life enjoyable. When I look back, I am always fulfilled with sweet memories.

Finally I want to thank my parents for their unconditional love and support.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. 7

LIST OF TABLES .............................................................. 14

ABSTRACT ................................................................. 15

CHAPTER 1 Introduction ...................................................... 16
  1.1 Cyber-Physical Systems ............................................. 16
  1.2 Embedded Human Systems .......................................... 17
  1.3 System Analysis and Verification .................................. 18
  1.4 Contributions ....................................................... 18

CHAPTER 2 Background ..................................................... 20
  2.1 Software Design of Cyber-Physical System ......................... 20
  2.2 Software Design of Embedded Human Systems ................... 22
  2.3 Design of Safe System Software ................................... 24
  2.4 Abstraction of Hybrid Systems .................................... 25
  2.5 Checking Safety with Analytical Methods ....................... 27
    2.5.1 Symbolic Reachability ....................................... 27
    2.5.2 Model Verification Tools ..................................... 28
  2.6 Safety Check with Numerical Methods ............................ 28
    2.6.1 Viable Capture Basin ....................................... 28
    2.6.2 Barrier Certificates ....................................... 30
    2.6.3 Level Set Methods .......................................... 32
    2.6.4 Stochastic Reachability .................................... 33
  2.7 Domain Specific Modeling ......................................... 33

CHAPTER 3 Methods ........................................................ 37
  3.1 Safety Validation ................................................... 37
    3.1.1 Difficulties .................................................. 37
    3.1.2 Abstraction of Embedded Human System ..................... 39
    3.1.3 Backward Reachable Set and its Properties ................ 40
    3.1.4 Multi-mode Reachability .................................... 43
  3.2 Parallel Computation Scheduling Algorithms ..................... 44
    3.2.1 Verification Properties .................................... 45
    3.2.2 Discovery of Time for Reachable Set Growth ............... 45
TABLE OF CONTENTS – Continued

3.2.3 Grid Variations .................................................. 48
3.2.4 Parallelization .................................................. 49
3.2.5 Single Target with Single Node ............................ 51
3.2.6 Single Target, Multiple Computing Nodes .............. 52
3.2.7 Multiple Targets with Multi-node .......................... 54
3.2.8 Variations of Group Scheduling ............................ 55

3.3 Safety Verification Software: Design ......................... 55
3.3.1 Objective ....................................................... 55
3.3.2 Software Requirements ....................................... 56
3.3.3 Application Domain ........................................... 56

3.4 Domain Specific Modeling ....................................... 57
3.4.1 Model-Based Software Synthesis ............................ 57
3.4.2 High Level Design .............................................. 58
3.4.3 Meta-model ..................................................... 60
3.4.4 Constraints ..................................................... 65
3.4.5 Model Design ................................................... 66
3.4.6 Code Generation ................................................ 68

CHAPTER 4 Results .................................................... 87
4.1 Parallel Computation .............................................. 87
4.1.1 Single Target .................................................. 87
4.1.2 Multi-target .................................................... 89
4.2 Examples ........................................................... 90
4.2.1 1D Example .................................................... 92
4.2.2 2D Example .................................................... 96
4.2.3 3D Example 1 .................................................. 101
4.2.4 3D Example 2 .................................................. 106
4.2.5 Improve Model Design Base on Reachable Sets ....... 114
4.2.6 Computation Time Quantization .......................... 114

CHAPTER 5 Conclusion ................................................ 124
5.1 Contribution ....................................................... 124
5.2 Future Work ........................................................ 126

APPENDIX A Safety Check Model Interpreter source codes .......... 127

APPENDIX B Example visualization MATLAB scripts ............... 133

REFERENCES .......................................................... 167
LIST OF FIGURES

2.1 Standard design. ......................................................... 25
2.2 Safe design. ............................................................. 25
2.3 Iterative design with verification tool. ............................... 26
2.4 The GME meta-modeling environment, from the Part Browser at the top left corner, domain expert can specify aspect to edit. ......... 36

3.1 The fundamental structure of an Embedded Human System. ............ 39
3.2 A larger example of an embedded human system utilizing the pattern shown in Figure 3.1. .................................................. 40
3.3 The capture set of steady state $p_1$ at time $\tau_1$ and $\tau_2$. .......... 41
3.4 The growth of the reachable set as $t$ increases. Note that at $t = 7$, the target of the previous mode $T_{q_i}$ is fully contained in the capture set for $T_{q_{i+1}}$. ......................................................... 47
3.5 The capture set $G(\tau_{q_1}^*, q_1)$ for mode $q_1$ is calculated using a sparse grid (the solid line), and a dense grid (the dotted line). The dense grid calculation utilizes a smaller value of $\tau_{q_1}^*$ than that used in the sparse grid, in order to more tightly satisfy Property 3.2.1. ......................... 50
3.6 The computation algorithm for the computation of single target with single node. ...................................................... 51
3.7 The computation algorithm for the computation of single target with multiple node. ...................................................... 53
3.8 Example multi-mode system where the thin circle represents the stationary state and the thick circle represents the transition state. .... 57
3.9 Example model created in GME (Generic Modeling Environment). .. 58
3.10 Interaction of three main components for the safety verification software. ..................................................... 59
3.11 The Behavior Model class diagram of meta-model. ...................... 61
3.12 The Behavior Model attribute of meta-model. ........................... 61
3.13 The Behavior Model visualization of meta-model. ....................... 62
3.14 The Behavior Model visualization configuration mode of meta-model. 62
3.15 The StateTransition Model class diagram of meta-model. ............ 63
3.16 The ReachSetSetup Model class diagram of meta-model. ............. 64
3.17 The Controller Model class diagram of meta-model. .................... 64
3.18 Constraints can be added to the meta-model. .......................... 65
3.19 The constraint can be described by OCL (Object Constraint Language). ............................... 66
3.20 A warning window will pop out once the constraint is violated in model design. ....................................................... 66
3.21 The state transition aspect of modeling environment. .............. 67
3.22 The ControllerSetting aspect of modeling environment inside a TransitionMode object. ........................................ 68
3.23 The ControllerSetting aspect of modeling environment inside a Controller object. ............................................... 69
3.24 The behavior aspect of modeling environment. ........................... 70
3.25 The Reachable Set aspect of modeling environment. .................... 70
3.26 The UML diagram for ReachSet class, where only important attributes and methods are listed. ............................... 71
3.27 The UML diagram for RouteNode class. .................................. 72
3.28 The UML diagram for SafetyCheck class, where only important attributes and methods are listed. ............................... 72
3.29 **Use Case 1 Description.** The use case description for verification of a single platform. ............................................. 77
3.30 **Use Case 2 Description.** The use case description for reusing the working model for other controllers. ......................... 78
3.31 **Use Case 3 Description.** The use case description for reusing the working model for other platform verification. ............... 79
3.32 **Use Case 4 Description.** The use case description for reusing original model to restore transition protocols. ........................ 80
3.33 **Use Case 5 Description.** The use case description for verification of shared operation behavior for multiple platforms. ............ 82
3.34 The use case diagram for safety check tool. ............................. 82
3.35 The sequence diagram of code generation process. ..................... 83
3.36 The sequence diagram of computation and result visualization process. ............................................................................. 83
3.37 The sequence diagram of reusing a working model. ....................... 83
3.38 The sequence diagram of reusing a failed or partially failed model. .... 84
3.39 The sequence diagram of reusing model for different platforms. ....... 84
3.40 The activity diagram for single platform verification. ..................... 85
3.41 The activity diagram for multiple platform and multiple controller verification. ................................................................. 86
4.1 Single target, with multiple available computing nodes. ............... 88
4.2 Single target, single computer with multiple cores. ...................... 88
4.3 Computation time of single target when there are multiple cores and nodes. (see Figure 4.1, Figure 4.2) .............................. 89
4.4 Results of 4 targets with 2 nodes. ................................. 90
4.5 Results of 4 targets with 4 nodes. ................................. 90
4.6 Results of 4 targets with 8 nodes. ................................. 91
4.7 Computation time of 4 targets when there are multiple cores and nodes.(see Figure 4.4, Figure 4.5 and Figure 4.6) .................. 91
4.8 GME model for 1D example. Using the same model as in Figure 3.9, the system can be built with modifications to the States, TransitionMode and system behavior descriptions. The location information can be added to the State block, different controllers and their parameters shall be added to each TransitionMode and output parameters such as available number of cores for computation and output folder, system dynamic equations, Hamiltonian and derivative equations are also provided to the system behavior descriptions at the bottom right corner of the model builder. ................................. 94
4.9 Minimum capture sets of transition starts from State 1 for two 1D person models. As the maximum backward velocity of Person 1(-0.5m/s) is lower than Person 2(-1m/s), the minimum reach time of Person 1(33.98sec) is longer than Person 2(23.72sec). Due to the higher maximum forward velocity(2m/s for Person 1 and 1.5m/s for Person 2) and longer minimum reach time, the minimum capture set of Person 1 contains more area than Person 2. .................. 95
4.10 Minimum capture sets of transition starts from State 2 for two 1D person models. The top figure shows the minimum capture set for Target 1, the bottom figure shows the minimum capture set for Target 2. Target 1 locates at left side of initial state and Target 2 locates at the right side of initial state. Due to the velocity differences in forward and backward directions, Person 1(22.50sec) has longer minimum reach time for Target 1 than Person 2(17.64sec), Person 2(17.92sec) has similar minimum reach time for Target 2 than Person 1(17.93sec). Behaviors for different decisions can be observed with minimum reach time and the shape of capture sets. .................. 97
4.11 Minimum capture sets of transition starts from State 3 for two 1D person models. The shape of capture sets is similar as the shapes in transition from State 2 to Target 2 as in Figure 4.10, as these transitions implement the same controller and they both move forward to reach the target. .................. 98
4.12 GME model for 2D example. The construction of this 2D Person example can be simplified by reusing the GME model of 1D example. By modifying States, TransitionModes and system behavior descriptions, the interpreter can generate MATLAB scripts to verify the design.

4.13 Minimum capture sets of transition starts from State 1 for three 2D person models. The minimum capture sets of Person 1 and Person 2 have similar shapes, as they both implement the same control logic, due to their differences in velocity range, they have different minimum reach time (Person 1 has reach time 20.57sec, Person 2 has reach time 34.50sec). Person 3 has a completely different minimum capture set shape because of its special control logic, and Person 3 also has longer minimum reach time (42.36sec). As both control logics are symmetric about x axis and y axis, we can observe that the capture sets are also symmetric about x axis and y axis.

4.14 Minimum capture sets of transition starts from State 2 for three 2D person models. As this transition model is from reusing the previous example, there are also two target states for State 2. As target 1 is closer to initial state than target 2, the minimum reach time for target 1 is generally longer than for target 2.

4.15 Minimum capture sets of transition starts from State 3 for three 2D person models. The shapes of minimum capture sets are different from previous transitions starting from State 1 and State 2, as the minimum reach time for this transition (14.07sec for Person 1, 25.71sec for Person 2 and 28.50sec for Person 3) is shorter than transition from State 1 and transition from State 2 to target 1, but longer than transition from State 2 to target 2.

4.16 GME model for 3D car example. The modeling with 3D dynamics can also be simplified by reusing previous models. After modification of States, TransitionMode and system behavior descriptions, safety check scripts can be obtained through model interpretation.

4.17 Minimum capture set slices at $x_3 = 0$ of transition starts from State 1 for three 3D car models. The minimum reach time for Car 1 and Car 2 are 1.71sec and 1.55sec.
4.18 Minimum capture set slices at $x_3 = 0$ of transition starts from State 2 for three 3D car models. The minimum reach times for Car 1 and Car 2 in the top figure are both 1sec. The minimum reach times for Car 1 and Car 2 in the bottom figure are 1sec and 1.21sec. As the initial estimate of minimum reach time for all these transitions is 1sec, it can be inferred that the actual minimum reach times for both cars in top figure and Car 1 in bottom figure are all less than 1sec.

4.19 Minimum capture set slices at $x_3 = 0$ of transition starts from State 3 for three 3D car models. The minimum reach times for both cars are 1sec, we can infer that the actual minimum reach times for this transition are less than 1sec.

4.20 GME model for 3D UAV refueling example, this model is also built through model reusing.

4.21 Minimum capture set slices at $x_3 = 0$ of transition starts from State 1 with sparse and dense grids for UAV 1. Comparing with Step 2 capture sets, we can infer that the Step 1 capture set slice underestimates the actual capture set in all directions.

4.22 Minimum capture set slices at $x_3 = 0$ of transition starts from State 2 with sparse and dense grids for UAV 1. The Target 1 Step 1 capture set slice has some overestimation in certain directions, and the Target 2 Step 1 capture set slice have overestimation in most directions.

4.23 Minimum capture set slices at $x_3 = 0$ of transition starts from State 3 with sparse and dense grids for UAV 1. The Step 1 capture set slice here has overestimation towards the bottom right direction.

4.24 Minimum capture set slices of transition starts from State 1 at $x_3 = 0$ for three 3D UAV models. The minimum reach times for these three UAVs are 2.86sec, 3.93sec and 2.29sec.

4.25 Minimum capture set slices of transition starts from State 2 at $x_3 = 0$ for three 3D UAV models. The minimum reach times for these three UAVs in top figure are 1.53sec, 2.93sec and 1.07sec. The minimum reach times for these three UAVs are 1.64sec, 3.59sec and 1.07sec.

4.26 Minimum capture set slices of transition starts from State 3 at $x_3 = 0$ for three 3D UAV models. The minimum reach times for these three UAVs are 5.43sec, 5.64sec and 6.93sec.

4.27 The capture set and collision set slices of three different UAVs in TransitionMode1 at $x_3 = 0$.

4.28 The capture set and collision set slices of three different UAVs in TransitionMode2 at $x_3 = 0$. 
LIST OF FIGURES – Continued

4.29 The capture set and collision set slices of three different UAVs in TransitionMode4 at $x_3 = 0$. .......................................................... 118
4.30 The capture set and collision set slices of three different UAVs in TransitionMode3 at $x_3 = 0$. .......................................................... 118
4.31 The unsafe set slices for different escape maneuvers at $x_3 = 0$. ........ 119
4.32 The GME model for 3D car system with one transition. .................. 119
4.33 The capture set slices of transition1 at $x_3 = 0$. The minimum reach time is 10.43sec, which means this design violates the protocol. ........... 120
4.34 The GME model for 3D car system with three transitions. ................. 120
4.35 The capture set slices of transition1(from [5, 5] to [10, 25]) at $x_3 = 0$ for updated model. The minimum reach time for this transition is 6.07sec. .......................................................... 121
4.36 The capture set slices of transition2(from [10, 25] to [15, 35]) at $x_3 = 0$ for updated model. The minimum reach time for this transition is 5.88sec. .......................................................... 121
4.37 The capture set slices of transition3(from [15, 35] to [20, 45]) at $x_3 = 0$ for updated model. The minimum reach time for this transition is 5.88sec. .......................................................... 122
4.38 The total computation time for the same reachable set computation vs different grid number setting. ................................................. 122
4.39 In the case of initial underestimation, there will be a slight punish- ment in total computation time for parallel scheduling algorithm. ....... 123
4.40 In the case of overestimation, the parallel scheduling algorithm could save huge amount of computation time. .................................. 123

B.1 The User Interface for verification results. All safety guaranteed transition states are displayed on the interface, there are one transition mode(TransitionMode1(o)) and five escape modes(Turn right, Turn left, Speed up, Slow down and Quick escape). The Pause button is at the lower right conner, by pressing the Pause button all state will be paused and the state will continue once this button is pressed again. The collision sets for transition states and escape states and capture sets for transition states are shown by different color and line types. Solid red line and solid green line are moving trajectories of UAV and tanker. The blue shape is the capture set slices for TransitionMode1 and the its related collision set slice is shown by the magenta shape, the target state is shown with symbol o. Different types of red shapes are collision sets for five escape modes. The black shape shows the minimum separation set of the tanker. ... 134
B.2 The state transition diagram for result visualization program. The system will start from the initial steady state and end when there is no transition state available for the steady state the system is targeting at. ................. 135

B.3 The function calling graph for result visualization program. All model-dependent scripts that are inside the dashed line rectangle are generated by the interpreter. The data from reachable set computation are represented with an elliptical shape. All model-independent functions are placed outside the dashed line rectangle. .............. 135
LIST OF TABLES

4.1 Use Cases tested for different examples. . . . . . . . . . . . . . . . . . . 92
4.2 The specifications of three example UAV models. . . . . . . . . . . . 110
ABSTRACT

Cyber-Physical Systems (CPS) are involved with both physical processes and computational processes. The interaction of physical components and computational components makes it difficult to analyze, design and verify this type of systems. The problem becomes more complex when an input or decision must be initiated by a human. For this dissertation, Cyber-Physical Systems with a human operator in the loop are called Embedded Human Systems (EHS). To ensure the safety of EHS such as traffic control systems, space shuttle control systems, nuclear power plant control systems and so on, it is critically important for human operators to fully understand both physical and computational processes. However, humans are usually easily overwhelmed by concurrent information, so this assumption becomes intractable when it comes to complex EHS with timing constraints.

This dissertation proposes a domain specific modeling language that takes advantage of hybrid system abstraction to retain important system behaviors and automatically generates self-configured system verification software. The verification software effectively reduces the computation time through parallel scheduling algorithm, and if the computation process detects a violation of the design requirements, verification can be halted without wasting computation resources. The modeling environment also allows a user to conveniently set design constraints to avoid flaws early in the prototype process, and reuse the available model for a family of different platforms. Several verification results of different platforms are shown to demonstrate the efficiency and reusability of the modeling environment.
1.1 Cyber-Physical Systems

Due to the fast development of computer hardware in recent years, computation speed and communication bandwidth are dramatically improved, making it possible to maintain strong links between cyber world and physical world. These strong links are found in Cyber-Physical Systems, which are characterized by a deep coupling of computational and physical processes. There is usually a feedback loop between physical and computational components, such that they can influence each other. The physical process is intrinsically concurrent and its time elapse is inexorable [1]. Results in CPS will improve the design of medical systems, traffic control systems, advanced automotive systems, process control, energy conservation, avionics and critical infrastructure control and so on.

This deep coupling inevitably makes the system complex to design, analyze and verify, and traditional techniques are costly to apply (and difficult to understand) [1]. The modeling of CPS usually contains two distinct approaches: one is to retain the important behaviors of the system, thus we can get a high fidelity system; the other is to omit details to get an abstraction of the system [1]. When we model physical systems, it is inevitable to omit certain details of the system, so what we get is always an abstraction of the original system. However, one challenging part of CPS modeling is to leverage the strengths and weaknesses of these two approaches, where we only omit unnecessary details of the system, thus the important system behaviors can be efficiently expressed.

As the physical components of CPS are usually the combination of continuous and discrete processes, they can be modeled as hybrid systems. Since continuous systems are usually modeled as time-based models while discrete systems are mod-
eled as state models, hybrid systems work as a bridge between these two system types [1]. The combination of these two types of system models is widely used to model real-world systems [2].

1.2 Embedded Human Systems

An Embedded Human System is one where the (potentially remote) system has some behaviors performed by on-board algorithms, while some inputs and decisions must be made by a human. This requirement for a human to understand the state of the remote system (in order to make decisions) is complicated if there is communication latency associated with the integrated system. With the development of modern information technology, more and more Cyber-Physical Systems require real time decision inputs from human operators embedded into the system [3]. The human decision may be safety critical to a wide range of embedded human systems, such as the air traffic control systems [4], ship and space shuttle control systems [5] [6], nuclear power plant control systems [7] [8] [9], highway traffic control systems [10], flexible manufacturing systems [11], medical care monitoring systems [12] [13] [14] and military defense systems [15].

In these safety critical systems, the interactions between human operator and the CPS could greatly improve or reduce the reliability and safety of the system. For example, the automation system of modern aircraft usually contains many layers of control with respect to a series of different maneuvers and different types of control input and protection techniques. However, flight management systems are usually designed with expensive simulations and standalone simulators for training uses (i.e., pilots) to avoid specific known problems. As it is impossible to verify the whole flight state space [16], certain maneuvers that are considered safe by the pilot may actually be unsafe in practical use. Such failures may be caused by the pilot’s confusion about current mode or next mode or the poor design of the system interface. For systems of this type, improvements in both system modeling and system design are proposed in this dissertation to ease the EHS development and
assist the embedded human operator.

1.3 System Analysis and Verification

In the application domain of Embedded Human Systems, almost every system should be designed to meet certain requirements. When it comes to safety critical systems, it is extremely important to validate the system design to satisfy these requirements. These requirements are also considered as system specifications or properties. The analysis of these systems will have to obtain a precise description about system specifications or properties based on the dynamic behaviors of EHS [1]. Once the system model is available, verification methods should be able to ensure that the final design does not violate any system specification or property.

Many computation techniques are developed for verification. These techniques may provide mathematical guarantees about system safety, where the safety of the system is defined as always staying within a certain desired subset of the state space [16]. Due to the success of modeling physical world and verification computation with hybrid automata, it is practical to model the physical components of CPS with abstracted system states and transitions among them. One of the challenges in system verification is to obtain a simple abstraction that is sufficient to ensure the safety of system, as a simple abstraction may have a small state space and its computation burden is relatively small [1].

1.4 Contributions

Complex EHS such as flight management systems usually contain several modes and each mode may have different behaviors (e.g., constraints, dynamics, disturbance). Previous contributions are either difficult to configure and implement, or they are specifically designed for certain domain and can not be reused for other domain. The main motivation of this research is to develop a reusable modeling and verification tool for multi-mode dynamic systems whose dynamical behavior can be expressed with Hamilton-Jacobi equations. Features should also be efficiently added
or modified through the system development and implementation.

The contribution of this dissertation is to propose a domain-specific modeling tool which provides a modeling environment for a family of multi-mode dynamic systems and also automatically generates code that is capable of safety verification for common properties. The tool takes advantage of Domain Specific Modeling and a parallel scheduling algorithm to speed up the modeling and verification process.

The structure of this dissertation is as follows: Chapter 2 introduces research related to system safety verification; Chapter 3 proposes our approaches to check the safety of general dynamic systems; Chapter 4 demonstrates the capability of our safety check tool with parallel computation examples and several use cases; Chapter 5 discusses contributions and possible improvements and features that could be added to the current tool.
CHAPTER 2

Background

2.1 Software Design of Cyber-Physical System

When we design a simple program, the program is generally expected to be predictable and reliable. These behaviors become extremely important in the design of Cyber-Physical Systems. High fidelity will be difficult to ensure if the software fails to represent important behaviors of the system [1]. The system software could work well, however the characteristic behaviors of the system may still be poorly represented. For example, timing constraints [17] may not be well represented in software systems written in C; as timing is not a basic attribute of C, thus a program that is running normally could miss certain deadlines. These deadlines may be tightly related to the normal functioning of the whole system, because system components could have different behaviors under the dimension of timing.

The failure of system software is sometimes due to abstraction failure, which fails to represent important system behaviors (i.e., omitting important details) [1]. This problem becomes more difficult with a complicated system [17]. The design of the system abstraction is difficult because many high-level design environments hide timing and other constraints. For example even the simplest program may still fail to express essential behaviors of the system [17]. In order to make the software predictable and reliable, the design should move from high level down to the low level to carefully set many low level configurations. For instance, the software system design for a system with strong timing behavior will require lots of configurations about the thread management, semaphores, priorities and so on [17].

In the area of software engineering, predictable and reliable software can always be obtained, as long as there is no evidence showing that it is impossible or unnecessary to get it [17], thus softwares with desired behaviors can be specifically designed.
Programming languages such as Java and C/C++ separates the user from the lower levels of the system, such that desired features of the software can only be obtained through careful and innovative design of these levels [17]. For instance, timing feature is a key factor for component-based system software to have reliable behavior [18].

For a Cyber-Physical System, as the computation and physical processes have tight interactions, timing features should not be removed or hidden through the abstractions [17]. The CPS software system will require a predictable timing, concurrent programs strongly depend on time, even small time variations may cause significant influence to the system. Many traditional approaches implement worst case execution time methods to solve the system timing issue, however, those solutions are generally uncertain, the uncertainty rises up as the complexity of the system increases [19].

Researches in deterministic methods that can handle the unpredictable behaviors of the system are motivated by the need to realize the full potential of the CPS [20]. Many innovative approaches in several design aspects are already developed [1]. By adding constraints to existing programming languages, predicted performances can be consistently achieved. C-like languages with multithreading support such as Split-C [21] and Cilk [22] both provide straightforward management of multithreading compared against low-level threads. The Guava language [23] supports safer read write lock by adding constraints that limit muti-thread access of unsynchronized objects to Java. The SHIM language [24] provides more controllable thread interactions. These approaches do not sacrifice performance, but they still have deadlocks and lack timing semantics.

If sacrificing performance is not a main concern, there are many other feasible solutions. For example, an instruction set architecture can be extended to support low cost instructions with precise time control [25], realtime garbage collection algorithm can bound pause times for memory management [26], and time related semantics can be added to programming languages [27]. Domain specific software development [28] is an innovative research direction to develop domain specific software. There
are many applications implementing model based software development. Temporal dynamics can be represented with sufficient semantic space [29], system characteristic behaviors of general purpose computation can be represented through interface designs [30]. Model based design could also provide enough semantics for many behaviors that are not traditionally defined. For example, the ability to apply interface theory to component-based design [31] is one innovative system design pattern, behaviors such as causality [32], realtime resource consumption [33], time constraints [34], protocols [35], depletable resources [36] and other factors [37] may also be important to specific systems.

With new semantics focusing on interactions between activities, coordination languages are shown to be more efficient than programming language level modeling for a variety of systems. Two examples, a channel-based coordination model [38] and an implementation framework for software architectures [39] demonstrate the potential of the coordination paradigm. There are other promising approaches such as the actor-oriented design [40] and service-oriented design [41].

2.2 Software Design of Embedded Human Systems

Due to the limitations of Cyber-Physical Systems (the behavior of physical process could only be partially represented and the computational process could only handle predefined algorithms and data structures) [42], when an Embedded Human System is cyber-physical, the computational and physical processes of the system must be understood by the embedded human. Assuming some disturbance in the system (such as the latency, complex nonlinear behaviors and system error, etc.), the remote system may behave completely different from the embedded human’s expectation, if the embedded human does not understand the true state of the remote system. In this case, human decisions may result in unsafe behavior by the remote system, if it is designed in such a way that inputs from the human are obeyed without question.

As concurrency is a basic behavior of physical system, the computation process should also have this behavior. However, related research has already shown that
“humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among simple collections of partially ordered operations” [43].

Many technologies have already been studied to improve interactions between human and cyber-physical systems. Technologies for identification of risks and hazards within the system [44] [45] are developed. Some system failures are results of improper design, the system design technology may lack certain safety feature or the design technologies are not appropriately implemented. After huge disasters caused by system design, novel technologies are introduced to improve the design [46] [47] [5]. However, introducing new technology may also bring unexpected behaviors to system, for example, new technologies may increase problems rather than reduce them for human operators of industrial processes [48]. Problems may also be caused by improper response to a false alarm, failure to monitor the actual state, automation-induced error compounded by operator error and so on [49]. Experimental studies of human operator’s model and awareness of the flight management system [50] shows the two main issue in interactions between human and system are i) lacking awareness of the actual mode and ii) gaps in human operator’s understanding of the functionalities of the system automaton.

Due to the difficulty of concurrency processing, specific system abstractions shall be developed. For instance, Embedded Human Systems may be abstracted into a state-based model, when transitions between states are enabled either by the remote system, or by the embedded human. The main drawback of this abstraction is that the embedded human may send the wrong input or command to the system. This may be due to mode confusion (due to time delay or system complexity), or even due to operational error of pressing the wrong button. Regardless, to avoid unsafe behavior, the system design patterns could be constrained to limit behaviors to known-good behaviors, which are verified against validation constraints that are agreed to be safe or desired for all systems.
2.3 Design of Safe System Software

For embedded human systems, hazards are defined as a set of conditions that can lead system into accidents and risks are defined as the possibility of occurrence of hazard with the related accidents, and the worst loss associated with the accident [51], thus safe embedded human system software should be able to provide sufficient knowledge that support decision making for the human operator and eliminate the possibility of accidents. Safety verification of embedded human system usually includes identification of hazards, criticality assessment of hazards and methods to eliminate hazards [51]. In the early development of software for embedded human system, engineers usually create a list of possible hazards after hazard identification, thus the designer should try to avoid all hazards in the list. As the development goes further, verification methods such as subsystem hazard analysis (SSHA) [52] [53] or fault tree [54] [55] can be introduced. Subsystem hazard analysis can identify hazards related to the individual subsystem, and a fault tree is constructed with a logical diagram of event sequences that may cause accident [51].

One promising system error management approach is to separate system errors into two groups: human error and system error [56]. The human error is mainly caused by the uncertainty of human behavior, while the system error is mainly related to the working conditions of the operator. The system shall be designed to effectively provide enough but not overwhelming information to the operator to improve the system safety.

Two main types of design patterns are developed for safe system design as in Figure 2.2 [51]. The standard design pattern is as in Figure 2.1a. The safety constraints of the system are translated into system requirements, so the system development should not violate these system requirements. In standard design pattern, the system constraints do not directly relates the system design, in order to improve the system safety during the design process, the safe design pattern is as in Figure 2.2a. In Figure 2.2a, the system constraints are still translated into a couple of system requirements, however, the system constraints are directly bounding
the system design, and the system requirements are used to verify the design. The safe design pattern avoids all possible violations of system constraints, but does not take advantage of system requirements and verification results. We propose the iterative design pattern as in Figure 2.3, where the system safety requirements are translated into system design constraints and system verification requirements, as some constraints can directly bound system design while other constraints can only be checked through model verification, thus verification required constraints are grouped as system verification requirements, after system design, system verification requirements are used to verify the design, if the design is not valid, we iteratively go back to system design and verify it until all verification requirements are satisfied.

2.4 Abstraction of Hybrid Systems

The research of safety verification to hybrid systems belongs to the intersection of control theory and theoretical computer science. Many modern hybrid systems are
safety critical, to ensure the safe operation of such systems, specifications shall be
developed in order to constrain the system or even help the system design process.
The verification of large scale systems and systems with complex dynamics is rather
difficult, one possible approach is to convert the verification problem into decision
problem by abstraction.

While the abstraction could simplify the complexity of systems, one important
rule is that the abstraction should retain the properties and behaviors of the original
system. As the abstracted system and the original system share the same properties
and behaviors, the verification of desired property of the abstracted system and the
original system will be equivalent. Only when no property-preserving abstraction
could be found for the original system, a sufficient abstraction is allowed, where
the verification of the abstracted system is sufficient to the original system. A
collection of property-preserving abstractions is discussed in [57]. Depends on the
desired property, different methods are implemented to develop abstraction, such
as reachability based abstraction [16], n-th derivative based abstraction [58] and
simulation and bisimulation relationship-based abstraction [59].

The system abstraction also plays an important role in human interface design of
embedded human systems. In the aviation industry, many incidents and accidents
attributed to the pilot’s confusion about the current flight mode and/or the transition
to the next mode, also called mode confusion. In the case of mode confusion,
a confusing control interface design could result in aircraft’s stall, for example. The
information that should be provided by the user interface is discussed in [60]. The
designer should pay enough attention to the information that is selected from the
underlying human-automation system, because the information will be used by the
user to control the system. In order to ease the difficulty of control when the user
is in *mode confusion*, it is suggested to provide a minimum set of information to
the user interface, so the user can safely complete the maneuver. As the problem
addressed by this research is to model and verify multi-mode dynamical system in
an effective and reusable way, the discussion of simplification of user interface is
outside the scope of this dissertation.

2.5 Checking Safety with Analytical Methods

Due to the complex behaviors of the physical environment, the integration of phys-
ical process and computation component shall provide enough information to assist
the embedded human make decisions or even avoid design errors early in the system
design process. Many approaches are developed for checking the safety of dynamic
systems with highly nonlinear behaviors.

2.5.1 Symbolic Reachability

In order to merge physical processes with information systems, the analysis and
design of embedded system is becoming more and more important. For example,
robots, UAVs and other autonomous vehicles will need this type of control for the
system, which contains both continuous and discrete components. Hybrid systems
are suitable for the modeling of this type of embedded system, which can be de-
scribed with both discrete event systems and differential equations. One important
problem in hybrid system control is the reachability problem. To make the control
safe, the system has to be kept away from unsafe regions of the state space. Al-
though lots of computer-aided verification tools have been developed for the model
checking and automated theorem proving, due to the decidability issue about com-
puting reachable sets of differential equations, the computing of reachable set is still very difficult.

The quantifier elimination method can be used to simplify the linear systems for the later computation. Linear systems of the form $\dot{\xi} = A\xi + Bu$ are shown to be decidable, and the computation of the exact reachable set for a family of linear vector fields can be obtained with the symbolic reachability computation [61]. However, it is still difficult to apply symbolic reachability to nonlinear systems.

2.5.2 Model Verification Tools

Symbolic model checkers [62] are developed for the verification of design parameters, however the dynamical properties of the system are still difficult to analyze when the system dynamics contain multi-mode or nonlinear behaviors. There are tools developed to verify specific properties, for example, polyhedral properties of hybrid systems can be verified using CheckMate [63], which is a MATLAB-based tool that determines reachable states with computational techniques. However, this approach is restricted to the verification of polyhedral property.

2.6 Safety Check with Numerical Methods

2.6.1 Viable Capture Basin

As the physical world usually exhibits both continuous and discrete behavior, the definition of hybrid system is proposed to describe dynamic systems with both continuous and discrete properties. The continuous and discrete dynamic behaviors can be described with a constrained differential inclusion and a constrained difference inclusion in a general hybrid system model [64][65]:

$$\dot{x} \in F(x), \ x \in C, \quad (2.1)$$

$$x^+ \in G(x), \ x \in D. \quad (2.2)$$

Suppose the hybrid system in the form of 3.1.4, 3.1.4 is represented with $H, F$ is called flow map, $C$ is called flow set, $G$ is called jump map, and $D$ is called jump
set. The definition for hybrid kernel is as following (from [66]):

**Definition 2.6.1** The hybrid (resp. strict hybrid) kernel of \( K \) for the impulse system \((F, \Phi, K)\) is the largest subset of initial states belonging to \( K \) from which starts at least one (resp. strict) hybrid viable solution. We denote this set \( \text{Hyb}_{(F, \Phi)}(K) \) (resp. \( \tilde{\text{Hyb}}_{(F, \Phi)}(K) \)).

The continuous system of hybrid system for the Viable Capture Basin is as following (from [66]):

**Assumption 2.6.1** (Continuous Set) \( C \) is compact and \( \Phi: C \rightarrow X \) is upper semi-continuous with compact values:

\[
0 < \delta_{\inf} \leq \inf_{x \in C} \inf_{y \in \Phi(x)} d(x, y) \leq \sup_{x \in C} \sup_{y \in \Phi(x)} \leq \delta_{\sup}
\]

so that \( \Phi \) has no fix point and its graph is closed. We set \( \forall x \notin C, \Phi(x) = \emptyset \)

**Assumption 2.6.2** (Continuous Map) \( K \) is a compact set and \( F \) is a Marchaud map satisfying \( \sup_{x \in K} \sup_{y \in F(x)} ||y|| \leq M \). This assumption implies that \( F \) is closed. As usual in the context of set-valued numerical analysis we need to consider “good” approximations \( F_\rho \) of \( F \) in the sense that \( \text{Graph}(F_\rho) \) remains in a not too large neighborhood of \( F \)

\[
i) \quad \text{Graph}(F_\rho) \subset \text{Graph}(F) + \Phi(\rho)B_{X \times X}
\]
\[
ii) \quad \bigcup_{x' \in B(x, \rho M)} F(x') \subset F_\rho(x)
\]

where \( B_{X \times X} \) denotes the unit ball of \( X \times X \) and \( \Phi \) goes to zero when \( \rho \).

Based on the assumption 2.6.1 and 2.6.2 about the flow and jump functions of hybrid systems, the following theorems (from [66]) are proposed:

**Theorem 2.6.1** (Viable Capture Basin) Let us assume assumption 2.6.1 and 2.6.2. Then

\( \text{Hyb}_{(F, \Phi)}(K) \) is the largest closed set of initial points for which there exists at least one hybrid solution viable in \( K \) which either reaches \( T \) in finite time or remains in the hybrid kernel for \( F \) of \( K \setminus T \): \( \text{Hyb}_{(F, \Phi)}(K \setminus T) \).
Theorem 2.6.2 (Minimal Time Function) Let us assume assumption 2.6.1 and 2.6.2. Then
\[ H_{\text{Hyb}}(\Psi_T,\Xi)(K \times R^+) \] is the largest closed set of initial points \((x_0, z_0)\) for which there exists at least one hybrid solution \(x(t)\) viable in \(K\) until it reaches \(T\) in a finite time \(\tau \leq z_0\). Moreover, \(V(x_0) = \min z_0 |(x_0, z_0) \in H_{\text{Hyb}}(\Psi_T,\Xi)(K \times R^+)\) is the Minimal Time-to-reach function which is the smallest positive lower semicontinuous supersolution of the following HJB equation
\[
\forall x \in \text{Dom}(V) \setminus C, \max_{u \in U} \langle f(x, u), -\frac{d}{dx}V(x) \rangle - 1 = 0
\] (2.6)

Theorem 2.6.1 defines Viable Capture Basin as the union of the initial conditions that could end in the target zone and the region that the hybrid solution remains in. Based on Theorem 2.6.1, Theorem 2.6.2 defines the Minimal Time Function which can be used to compute the initial conditions that can reach the target zone within minimum amount of time. The Viable Capture Basin approach takes advantage of HJB formulations that could check whether the initial condition could reach the target zone within finite time. However we still need to solve the HJB equation in order to obtain a numerical solution for the Minimal Time Function.

2.6.2 Barrier Certificates

The Barrier Certificate [67] is a function of state which proves that all trajectories of a hybrid system will not enter the unsafe zone. The zero level set of the Barrier Certificate is used to separate the initial conditions of those trajectories that end in unsafe zone. Suppose there is a continuous system
\[
\dot{x} = f(x, d)
\] (2.7)

The Barrier Certificate function(from [67]) is defined as following:

Definition 2.6.2 Barrier Certificate

Let the system 2.7 and the sets \(X, D, X_0\) and \(X_u\) be given. Suppose there exists a barrier certificate, namely a function \(B: X \rightarrow \mathbb{R}\) that is differentiable with respect to its argument and satisfies the following conditions:
\[ B(x) > 0 \quad \forall (x) \in \mathcal{X}_{u}, \]  
(2.8)

\[ B(x) \leq 0 \quad \forall (x) \in \mathcal{X}_{0}, \]  
(2.9)

\[ \frac{\partial B}{\partial x}(x) f(x, d) \leq 0 \quad \forall (x, d) \in \mathcal{X} \times D \text{ such that } B(x) = 0, \]  
(2.10)

then the safety of the system 2.7 is guaranteed. That is, there exist no trajectory of the system 2.7 contained in \( \mathcal{X} \) that starts from an initial state in \( \mathcal{X}_{0} \) and reaches another state in \( \mathcal{X}_{u} \).

One advantage of this method is that the construction of Barrier Certificates does not need explicit computation of reachable sets, even when dynamics has non-linear, uncertain behaviors or other constraints. The following proposition (from [67]) proposes the approach that convert the computation of Barrier Certificate to the sum of squares optimization problem.

**Definition 2.6.3** Sum of Squares Formulation Let the hybrid system \( H \) and the descriptions of all the sets \( I(l), D(l), \) Init\( (l), \) Unsafe\( (l), \) Guard\( (l,l'), \) and Reset\( (l,l')(x) \) be given. Suppose there exist polynomials \( B_{l}(x) \) and \( \lambda B_{l}(x, d), \) a positive number \( \epsilon, \) and vectors of sums of squares \( \sigma_{\text{Unsafe}(l)}(x), \sigma_{\text{Init}(l)}(x), \sigma_{I(l)}(x, d), \sigma_{D(l)}(x, d), \sigma_{\text{Guard}(l,l')}(x, x'), \sigma_{\text{Reset}(l,l')(x, x')}, \) and \( \sigma_{l,l'}(x, x') \), such that the following expressions:

\[ B_{l}(x) - \epsilon - \sigma_{\text{Unsafe}(l)}^{T}(x) g_{\text{Unsafe}(l)}(x) \]  
(2.11)

\[ - B_{l}(x) - \sigma_{\text{Init}(l)}^{T}(x) g_{\text{Init}(l)}(x) \]  
(2.12)

\[ - \frac{\partial B_{l}}{\partial x} f(x, d) - \sigma_{D(l)}^{T}(x, d) g_{D(l)}(d) - \sigma_{I(l)}^{T}(x, d) g_{I(l)}(x) - \lambda B_{l}(x, d) B_{l}(x) \]  
(2.13)

\[ - B_{l'}(x') + \sigma_{l,l'}(x, x') B_{l}(x) - \sigma_{\text{Guard}(l,l')}(x, x') g_{\text{Guard}(l,l')}(x) \cdots \]  
(2.14)

are sums of squares for each \( l \in L \) and \( (l, l') \in L^2, \ l' \neq l \). Then \( \{B_{l}(x)\} \) satisfies the conditions in Theorem 2 [67], and therefore the safety of the system is guaranteed.
The only limitation of this approach is that the continuous dynamics of hybrid systems should be described by polynomial vector fields and the invariant sets, guard sets, etc of the hybrid systems should also be described by polynomial equalities and inequalities.

2.6.3 Level Set Methods

The objective of reachable set research is to find the set of initial conditions that could reach a pre-defined subset of continuous space under continuous time dynamics. This research could effectively improve the analysis and verification of continuous time nonlinear systems or even hybrid systems.

The computation tools for reachable sets of discrete state space are already developed [68]. To solve the computation problem of continuous state space reachable sets whose dimension exceeds four or five degrees of freedom. Based on level set methods [69] [70] [71] and viscosity solutions [72] [73], methods for backward reachable sets computation [74] [75] are developed. The time-dependent Hamilton-Jacobi based reachability computation [76] has a strong advantage when there are inputs and uncertain parameters within the nonlinear system. The time-dependent Hamilton-Jacobi based reachability computation also generally has better accuracy than other formulations with discontinuous solutions, because the solutions of the time-dependent formulation is continuous and defined for the whole state space. Comparing with computation methods based on static Hamilton-Jacobi equations [77] [78] and viability theory [79], the time-dependent computation method has desired efficiency and accuracy. As the Hamilton-Jacobi based reachability has a strong connection with hybrid systems, it has been successfully implemented in many applications in a hybrid system verification.

The disadvantage of computation methods is that computation cost grows exponentially in number of dimensions. There are a series of more effective computation methods [80] [81] [82] for reachable set computation, however only specific types of dynamical system and reachable sets can take advantage of these approaches.

Based on reachability computation, complex nonlinear dynamic systems can be
decomposed into a series of discrete abstractions [83] [84] [82] [85] [86] [87] [88] [89] [58] [90], thus safety protocols can be developed to avoid unsafe states. These abstractions could also simplify the human-automation interface for flight management system or automatic landing, so that the pilot could be provided with enough information without information overloading. The reachable set is also useful in the design of controllers. To ensure the safety and target reachability of sampled-data switched systems, a robust reach-avoid controller can be obtained using the reachable set as the system feedback [91]. The reachability computation could also be used with optimal control to obtain the maximum controllable invariant sets [92]. Computation methods are also effective in verification of controller designs [93] [94].

2.6.4 Stochastic Reachability

The solutions of stochastic differential equations can be approximated with Markov chain, using this approximation, the probability of the system trajectory to enter a certain target set within possibly infinite-look-ahead time can be estimated [95]. In many safety critical applications, events that may put a system at risk are not common, so probability methods such as Monte Carlo simulations can be used to estimate the probability of these events and thus provide a certain level of confidence to verify or certify the system. However, if the environmental disturbances are under certain bounds, such as relevant disturbances of speed and turning rate, deterministic methods could provide stronger safety guarantees.

2.7 Domain Specific Modeling

When the size of Cyber-Physical System becomes larger, it will contain more strong combinations of physical components and information components. As the environment variables (such as the system requirements, disturbances and so on) are always changing, thus only reconfigurable software are developed to satisfy requirements effectively. After analysis of the system, a rigorous model is expected to be developed. In order to reuse the model, the model must also be general enough. However, as
the environment of actual implementation may have huge differences, so the model built by descriptive tools could only be applied to certain specific systems. For example, a tool developed to describe process control may not be used to describe a flight management system, as these systems may have different requirements and constraints. The process control may need precise control of the system state, while the flight management system may only try to avoid cases that cause collision. The difficulties in describing these different types of systems can be transformed into problems in domain specific modeling.

To solve the problem of system description, a meta-level model that does not describe the model, but describes the language used to describe the model can be developed. Once the meta-level language is available, the domain language can be obtained through interpretation. Ontology, syntax and interpretation are three fundamental elements of meta language. Ontology generally means that domain specific entities and relationships among them can be represented with a shared definition, syntax at the same time defines principles of model construction in specific domain, and interpretation is the mapping between different domains. In application, ontology and syntax describes the domain language, while interpretation generates code for the specific domain. One benefit of implementing meta language is that model evolution and rapid configuration can be effectively completed. However, the evolution is still not trivial, as the new meta-model should be compatible with the old model, such that the model built in the new environment should not conflict with the model built in the old environment without migration of the old model [96]. From the hybrid system evolution example in [97], semantic model evolution is much more complex than syntactic model evolution.

The idea of domain specific modeling focuses on the implementation of a specific engineering domain. A meta-model is used by DSM to define the description language and constraints of the model. The modeling environment can be automatically generated from the meta-model using meta interpreter defined by the user. Once the model is built under the modeling environment, it can be interpreted automatically into the target application. Once the meta-model is available, a family
of systems can be quickly and effectively built by taking advantage of the reusable components. With domain specific modeling, extensible models can be developed. For example, the Activity Modeling Tool (AMT) [98] is developed to connect to a real-time database in order to monitor and simulate the process industry. The Hybrid Systems Interchange Format (HSIF) [99] is developed to generate a common repository for hybrid systems with model-based representation of mathematics. GME (Generic Modeling Environment) [100] is a DSM tool which provides a program synthesis environment. GME has many other useful features, such as paradigm generation with formal modeling environment specifications, the techniques of the modeling environment includes hierarchy, sets, reference and explicit constraints.

In GME, MGA (Multi-Graph Architecture) defines a toolbox for the Model Based Software Synthesis. The following objects are the most important representation of concepts in the modeling environment:

- **Model** Model contains three attributes: state, identity and behavior. It also allows the objects such as Atom, Model, Reference, Set and Connection to be its parts.

- **Atom** Atom represents simple object which has no internal structure, it is used to represent entities that can not be break up into parts.

- **Reference** Reference works as pointers in a couple of programming languages.

- **Connection** Relationships between different parts are expressed with Connections. Connection is expressed with a line with an arrow, it has two important attributes: appearance and directionality. The appearance represents the type of connection between different objects, while directionality represents the direction information of the connection which can be distinguished by the arrow of the line.

- **FCO** FCO is First Class Object, which can be inherited by different objects (Atom, Reference, Model, etc.).
Figure 2.4: The GME meta-modeling environment, from the Part Browser at the top left corner, domain expert can specify aspect to edit.

To reduce the information displayed in the interface, the GME meta-modeling environment as in Figure 2.4 is separated into four design aspects: Attributes, ClassDiagram, Visualization and Constraints, thus the design space is decomposed into four different parts. The Attributes Aspect allows user to edit field of objects, the ClassDiagram Aspect allows user to create objects and relationships among them, the Visualization Aspect allows user to set the object that will be displayed to the modeler in modeling environment(some object will be hidden from the modeler), the Constraints Aspect is used to set constraints to objects with OCL.
CHAPTER 3

Methods

3.1 Safety Validation

3.1.1 Difficulties

As the large scale embedded human system usually contains nonlinear dynamics, the reachability verification of the model is thus nontrivial: it is either difficult to find a method to handle the nonlinearity in reachability analysis, or the computation is both expensive in time and memory space.

Besides the nonlinearity, an embedded human system may contain a series of different operations, and each operation may have its specific dynamics or constraints, such as the transmission of the car has several different gear-ratios. Thus multi-mode controllers should be applied to the model, which also increases the difficulty of system validation.

Another important feature of system validation is the reusability of the model. This feature should be considered during the design process to make the results of system validation not only useful for certain model, but also to a family of models with different specifications.

This dissertation approaches the issue of safety in terms of avoiding an area of the continuous state space which is considered unsafe. Many other research tries to solve the verification problem of a multi-mode dynamical system, however, there is not a systematical way to approach it. The situation usually becomes worse when the dynamical system has arbitrarily complex dynamics. This dissertation proposes a systematical way to model and verify multi-mode complex dynamical system as long as the system behavior can be represented with Hamilton-Jacobi equations. For a continuous system, the application of Hamilton-Jacobi reachability can be used. This approach considers a pre-specified subset of the continuous state space and
examines how this space changes under continuous time dynamics. Many calculation techniques of the Hamilton-Jacobi reachability can be found in the work of Mitchell [101] and has been applied by Tomlin et al. to many applications [84][83].

This approach to reachability is widely used in the analyzing and verifying continuous time nonlinear systems, and the previously cited papers provide a firm background upon which to justify these approaches. For readability, a brief reiteration of some definitions necessary to utilize those techniques is provided in the next section.

If we set the initial condition of a reachability calculation to be some state that we want to avoid, then continuous reachability can tell whether the system can stay out of the unwanted state: thus, a verification can be performed. The validation point is that all states should avoid some initial conditions; the verification point is whether or not a specific controller satisfies the validation criteria. In aeronautical applications, this type of verification may be briefly called a safety check, which permits model checking for autonomous aerial refueling [102] and conflict resolution maneuvers [103].

The system to be examined for safety properties in this dissertation is an embedded human system. The embedded human system is considered to be characterized by the fact that it is multi-modal and that a human controller can initiate some state transitions, while the remote system can also initiate some transitions. Clearly, complexities can occur if both the embedded human and remote system can initiate the same transitions between states.

In traditional information technology applications not characterized as embedded human, transitions of a multi-modal system (e.g., a mobile phone app) may be initiated by either the system itself or by a human, but there may be some state transitions that are only initiated either by the human or the system. For the embedded human systems in this paper, the delay in communication between the human and the physical system affects the safety and efficiency of the system. As the remote system becomes more complex, and difficult to estimate state without feedback, the embedded human is more likely to make faulty decisions due to un-
certainty. In cases such as these, it is desired to design the system such that safety of execution is explicitly guaranteed, even if it is not explicitly designed.

3.1.2 Abstraction of Embedded Human System

In order to design the system such that its safe behavior can be evaluated more easily, some key abstractions are proposed for an embedded human system’s design. Consider Fig. 3.1, which shows a basic state transition model of an embedded human system. In this figure, \( q_1 \) and \( q_2 \) represent system steady states, where the dynamics of the system are stabilized around an operating point. State \( p_1 \) represents a “Moving” state (shown in shorthand as a bold circle), which transitions the system between operating points. The transitions between different states are annotated by events and guard conditions.

Key in this abstraction is that to leave a mode at an operating point, the transition between \( q_1 \) and \( q_2 \) can only be initiated by a human controller. While moving, the system will execute some controller associated with mode \( p_1 \). Once the guard condition \( X \in T_{q_2} \) is satisfied, the system will transition to stationary mode \( q_2 \).

![Figure 3.1: The fundamental structure of an Embedded Human System.](image)

With this key abstraction, the complexity of an embedded human system can be scaled, as shown in Figure 3.2. It is clear that in each transition between operating points, that the arrival at an operating point is handled by the remote system’s evaluation of the system state, while departure from an operating point is evaluated by the embedded human controller.

However, it is not immediately clear (and in fact, not true) that any arbitrary implementation of controllers for each mode \( q_i, p_i \), would result in safe behavior simply through the constrained structure of such a system. Because this dissertation
assumes that the remote system is a cyber-physical one in which safety is determined by its location in continuous space, we utilize the reachability of the system modes in order to show safe behavior.

3.1.3 Backward Reachable Set and its Properties

One benefit of the approach in this paper is that the validation criteria for safe system execution can be inferred (and stated) without the need for an implementation. In other words, the design criteria for each mode $q_i, p_i$ of a system can be stated without either the system dynamics, or the modal controllers. The following section outlines what analysis can be inferred from system structure, and how system structure implies the validation criteria. First, some definitions (from the theory of continuous reachability) are necessary. These definitions are taken from \[83]\ in order to remain consistent with terminology and symbols used in the literature.

For each reachable set computation, we assume that there exists some system dynamics $\dot{x} = f(x, u, d)$ where $x$ is the state, $u$ is the control input (determined by some controller, $g(x, k)$, where $k$ is controller parameter), and $d$ is some disturbance.

**Definition 3.1.1** Let $T \in \mathbb{R}^n$ be called the target set, a subzero level set of a scalar of the function of states at time $t = 0$ for the time varying function $\phi(x, t) : \mathbb{R}^n \to \mathbb{R}$. 
Specifically,

$$T = \{ x \in \mathbb{R}^n, t = 0, \phi(x, t) \leq 0 \}$$

(3.1)

**Definition 3.1.2** The backwards reachable set is the set of $x$ for which there exists a control input signal $u(t)$, $t \in [-\tau, 0]$ such that regardless of the choice of the permissible disturbance $d(t)$ during that time period, that $x$ can be driven into the target, namely $x \subseteq T$, by time $t = 0$ (i.e., within time $\tau$).

The LSM approach, described in [101], applies the Hamilton-Jacobi equation such that the solution to a particular partial differential equation at some time $-\tau$ provides the values for $\phi(x, -\tau)$. Justification of this approach is provided in [76].

Two specific backward reachable sets are used in this work. The first describes driving the solution to a desired target, and the second describes regions of the state space that could lead to entering an undesired target.

**Definition 3.1.3** The capture set is the backward reachable set $G_T = \{ x \in \mathbb{R}^n, \phi(x, -\tau) \leq 0 \}$.

The capture set defines $x$ such that when applying $u(\cdot)$ for some time $\tau$, that $x \subseteq T$ after time $\tau$. The definition of viable capture basin is similar to the capture set, but the system belonging to the viable capture basin may reach the target set $T$ in finite time and remain in the capture basin for ever. Thus, a capture set is defined with more timing information compared to the viable capture basin.
Consider a target set named $M \in \mathbb{R}^n$, which represents an undesired area of the statespace.

**Definition 3.1.4** The collision set is the backward reachable set $G_\tau = \{ x \in \mathbb{R}^n, \psi(x, -\tau) \leq 0 \}$.

The collision set defines $x$ such that when applying $u(\cdot)$ (used to calculate the capture set for a target $T$), the system will enter $M$ within time $\tau$. Here a different time-varying function, $\psi(\cdot)$ is used to distinguish the different Hamiltonian required to specify the collision set’s growth. The definitions of the Hamiltonian functions are available from several references [104, 76, 83].

*Note:* For every capture set, there is a collision set defined for the same $f(\cdot)$, $g(\cdot)$, $T$, and $\tau$.

**Definition 3.1.5** The reach-avoid set $R \setminus A = \{ x \in \mathbb{R}^n, \forall t \in [-\tau, 0], \phi(x, t) \leq 0, \psi(x, t) > 0 \}$ is the backward reachable set under an admissible control policy.

The reach-avoid set [91] is the region where the system will enter the target set and always stay away from the collision set, which could be used to ensure the safe transition.

A graphical representation of these sets is shown in Figure 3.3. The capture set of $q_2$ at time $\tau_1$ and $\tau_2$ are shown as $G_{\tau_1,q_2}$ and $G_{\tau_2,q_2}$. As $\tau_1 \leq \tau_2$, then according to Definition 3.1.3, $G_{\tau_1,q_2} \subseteq G_{\tau_2,q_2}$.

An additional visual feature to note is that $T_{q_1} \notin G_{\tau_1,q_2}$. Thus, it is not the case that within time $\tau_1$ that the system can transition from the continuous region defined by $T_{q_1}$ to that defined by $T_{q_2}$. However, since $T_{q_1} \in G_{\tau_2,q_2}$, this means that the system can transition within time $\tau_2$.

With these definitions of the reachable set, we can state that in order to be a valid system, that the transition from $q_1$ to $q_2$ must happen in some bounded time (where that time is defined as a design parameter). An additional property to be a valid design is that regions identified in the collision set should not include regions of a previous steady state. In order to verify a system’s validity, the set must be
computed; however, in order to state the validation criteria, only the structure of the system is required.

In the presented approach, constraints on the system design are what permit the validation criteria to be extracted from the structure of the model. For instance, as in the transition model in Fig. 3.1, we can learn that there is always a transition state between two steady states to perform the transition. Since the transition state should listen to the command initiated by the human to transition, it is this state that must contain the controller with suitable parameters for this transition. Once the guard condition to leave the moving mode is satisfied, transition will take place to a steady state. Such a model has the following structural properties, then:

**Property 3.1.1 steady state to moving state.** A steady state may transition to one or more moving states, but not to another steady state, and transition can occur based only on input from the human controller;

**Property 3.1.2 moving state to steady state.** A moving state may transition to exactly one steady state, and only when a guard condition is satisfied;

**Property 3.1.3 moving state constraint.** Each moving state can be entered from exactly one steady state.

**Property 3.1.4 transition validation.** If the previous steady state is contained within the capture set of the next steady state without exceeding the boundary transition time, the transition from previous steady state to next steady state is valid.

Property 3.1.3 in the above list is a strict requirement that may be relaxed in future papers as it adds to the size of the state model diagram. Property 3.1.2, though it may seem somewhat rigid, prevents ambiguity in transition from moving to steady modes.

3.1.4 Multi-mode Reachability

A general hybrid system defined in Equation and can also be written as $\mathcal{H} = \langle F, C, G, D \rangle$ where $F$ is the continuous dynamics, $C$ is the continuous set, $G$ is
the discrete dynamics and $D$ is the discrete set.

Consider a multi-mode hybrid system that shares the same set $M$, but varies $T$ and $g(\cdot)$ for each mode. The hybrid system, $\mathcal{H}$, is defined as a tuple,

$$\mathcal{H} = \langle Q, X, V, f, E, G, T, q_0, T_0 \rangle$$

(3.2)

where $Q = q_1, \ldots, q_N$ is the set of discrete modes, $X$ is the continuous state space, $V$ is the set of continuous input variables, $f$ describes the continuous dynamics of each mode, $E$ is the set of transitions (edges) between discrete modes, $G$ is the set of controllers, $M$ is the set of $X$ to be avoided, $T$ is a set of target regions within $X$, $q_0$ is the initial mode, and $T_0 \subseteq X$ is a region in which the initial state is guaranteed to be. The set of transitions is further refined as $e \in E, e = \langle q_1, \gamma, q_2 \rangle$, where $\gamma$ is a guard condition that permits switching from mode $q_1$ to $q_2$ iff $\gamma \rightarrow true$.

The system dynamics $\dot{x} = f$ and $M$ are considered to be equivalent for each control mode. The resulting hybrid system has states $q_i \in Q$, and the $i_{th}$ control mode has a target ($T_{q_i}$) and controller ($g_{q_i}(\cdot)$).

Therefore each mode has a capture set $G_{\tau_{q_i}, q_i}$, and a collision set $G_{\tau_{q_i}, q_i}$. It is important that the $\tau_{q_i}$ match for the capture and collision sets with the same index, as the reachable sets are used as a decision tool to determine whether or not switching between modes is possible. For example, one such use of these reachable sets is as guard conditions for transitions between control modes. As shown in Figure 3.2, these guard conditions permit switching modes if the state is within the capture set, and not within the collision set, of the next control mode.

3.2 Parallel Computation Scheduling Algorithms

The approach involves verifying the properties of the multi-mode controller while (not after) computing the capture and collision reachable sets in parallel. In order to fully explore the approach, the desired properties of the system to verify are given, followed by explanation of how the presented approach satisfies these criteria, while permitting parallel computation of the capture and collision sets. The contribution
of this scheduling algorithm is that:

i) The algorithm takes advantage of multi-node/core hardware to speed up the computation.

ii) The results from initial low cost prediction are used to optimize later high cost computation, the total computation time is reduced.

iii) Once system design error is found in one process, the whole computation will be halted and user will be notified.

3.2.1 Verification Properties

The following properties must be satisfied by a controller in this paper to be considered safe.

**Property 3.2.1 Capture/Target Subset.** For transition from a mode $q_i$ to $q_j$, the target of $q_i$ should be within the capture set of the target of $q_j$, i.e., $T_{q_i} \in G_{(\tau_{q_j}, q_j)}$.

For transition from a mode $q_i$ to $q_j$, the edge should be guarded to ensure the current continuous state is in the capture set for $q_j$. In order to fulfill this requirement while maintaining the ability for the system to execute, the following properties should hold.

**Property 3.2.2 Collision/Target Nonoverlap.** For transition from a mode $q_i$ to $q_j$, no portion of the target of $q_i$ should lie inside the collision set (i.e., unsafe reach set) of $q_j$, i.e., $T_{q_i} \cap G_{(\tau_{q_j}, q_j)} = \emptyset$.

**Property 3.2.3 Time Equivalence of Capture/Collision.** For the capture and collision sets used in Property 3.2.1 and Property 3.2.2, the time should be equivalent for compared sets.

3.2.2 Discovery of Time for Reachable Set Growth

To satisfy the above properties, the capture set computation and collision set computation must be computed using the same time (i.e., synchronized). If these two
sets are not synchronized, the comparison will not make sense. For example the system may start inside the capture set defined by $G_{(1.5s,q_i)}$ but outside the collision set defined by $G_{(1.2s,q_i)}$. In cases such as this, there is the possibility that the collision set, if grown for 0.3 more s, would include the state at which the decision is made.

In the reverse case (where the collision set is for a time larger than the capture set), the conservative estimates may invalidate certain solutions that would otherwise be considered to satisfy safety properties.

In this paper, it is assumed that the time to transition between the $T$ for each mode is unknown; therefore, it is necessary to discover the time for which to grow the reachable sets (i.e., to determine $\tau_{q_i}$ for the solutions $\phi(x,-\tau_{q_i})$ and $\psi(x,-\tau_{q_i})$) as part of the calculation of the reachable sets themselves.

Let the lower bound $\tau^*_{q_i}$ be the smallest time to satisfy Property 3.2.1 for transition into mode $q_i$ from the target of the previous mode. An initial estimate, $\tilde{\tau}_{q_i}$ is chosen for initial calculations of the capture and collision sets, giving the following three cases:

**Overestimation**

$\tilde{\tau}_{q_i} \gg \tau^*_{q_i}$. In this case, the transition time was drastically overestimated, and there is some time significantly less than $\tilde{\tau}_{q_i}$ that satisfies Property 3.2.1. Detecting this case is difficult, without adding additional logic as the reachset grows (slowing the computation), so if Property 3.2.1 is satisfied after the initial estimate of $\tilde{\tau}_{q_i}$, then the capture set is computed again using $\tilde{\tau}_{q_i}/2$; this process is repeated until the $\tau^*_{q_i}$ is sufficiently approximated.

**Underestimation**

If the estimation error $\tilde{\tau}_{q_i} - \tau^*_{q_i} < 0$, then at the end of the computation $T_{q_i}$ will not be completely included in $G_{(\tau_{q_i},q_j)}$. In this case detection is trivial, and the approach is to extend the computation until Property 3.2.1 is satisfied. Using the result of $G_{(\tilde{\tau}_{q_i},q_j)}$ as the initial condition for growing the reachset, it is trivial to allow the set
to grow for $\Delta \tau$ time longer, and check for satisfaction of Property 3.2.1.

It is tempting, therefore, to set the initial estimate of $\tau_{q_i}$ to be very small, and then to check whether or not Property 3.2.1 is satisfied in $\Delta \tau$ increments. However, this approach (although trivial to implement) consumes tremendous memory and other computing resources to carry out. As an example of this, when $|\tau^*_{q_i} - \tau_{q_i}|$ is 50s, and $\Delta \tau$ to be 0.001s, if number of time extensions between checks is 100, then about 500 checks will have to be processed by the end of the computation. The uncertainty of $\Delta \tau$ and $\tau^*_q$ makes this method difficult to apply, and for large grids, the check for Property 3.2.1 may take several seconds to carry out.

Figure 3.4 shows this process using $\Delta \tau = 1, \tau_{q_i} = 1$. Note that in this case, several iterations were required in order to satisfy Property 3.2.1.

![Figure 3.4: The growth of the reachable set as $t$ increases. Note that at $t = 7$, the target of the previous mode $T_{q_i}$ is fully contained in the capture set for $T_{q_i+1}$.](image)

**Sufficient approximation**

From the above discussion, it can be seen that the optimal case will be $\tau^*_{q_i} - \tau_{q_i} \approx \epsilon$, where $0 < \epsilon \leq \Delta \tau$. This requires an approximately optimal estimation of the actual minimum transition time, and a single time extension using a sufficiently small $\Delta \tau$. For example, in Figure 3.4, an initial estimate of $\tau_{q_i} = 6$ would be considered a good initial estimate, because only a single extension of $\Delta \tau = 1s$ is required to satisfy Property 3.2.1. Of course, with a smaller $\Delta \tau$, the value of the final
\(\tau^*_q\) will be smaller, but this requires significant computational resources unless the estimated lower bound is sufficiently approximate to the lower bound. With these understanding of the penalties of estimation, the following function to calculate the capture set is introduced.

**Definition 3.2.1** Let \(C_G(T_{q_i}, T_{q_i+1}, \bar{\tau}_{q_i+1})\) be called the capture set calculation function. Utilizing the provided estimate \(\bar{\tau}\), it calculates the capture set such that Property 3.2.1 is satisfied, and returns as its result the set \(G(\tau^*_{q_i+1}, q_{i+1})\) as well as the time \(\tau^*_{q_i+1}\), which it determined was the appropriate lower bound. \(C_G(\cdot)\) determines in its execution whether the \(\bar{\tau}\) is overestimated, underestimated, or sufficiently approximated, and returns the lowest bound, based on \(\Delta \tau\).

### 3.2.3 Grid Variations

All computational solutions utilizing the HJ equation require a discretization of the statespace in order to calculate the level set functions as time progresses. As discussed in [101] the computation time grows exponentially with the number of grid points. Thus, determination (and exploration) of the approximate times for each capture set, if done on a relatively sparse grid, will take less time than if performed on a relatively dense grid.

The first step is to set a relatively large value for the grid size \(\Delta x\), and use the algorithms described in Section 3.2.2 to estimate the transition time \(\tau^*_q\) for each mode. Figure 3.5 shows an example calculation for some mode \(q_1\), which will give a rough result of the capture set \(G(\tau^*_{q_1}, q_1)\), the surface outlined by solid lines. Because the determination of \(\tau^*_q\) is performed using the sparse grid, the computation can be performed relatively fast, compared to using a dense grid.

Second, the grid is refined in order to provide more points at which to estimate the growth of the capture and collision sets. This is performed by decreasing the grid size to \(\Delta x/N\) (where \(N\) is chosen by the system designer) in order to provide a more accurate representation of the surface of \(G(\tau^*_{q_1}, q_1)\), which is shown as a dotted line in Figure 3.5.
Suppose the first step is using grid size $\Delta x_1 = \Delta x$, the second step is using $\Delta x_2 = \Delta x/N$, and the result estimate from first step is $\tilde{\tau}_{(q_i, \Delta x_1)}$, the second step gets $\tilde{\tau}_{(q_i, \Delta x_2)}$. Although the sign of error of these estimates is uncertain, we can prove that $|\tilde{\tau}_{(q_i, \Delta x_2)} - \tau^*_q| \leq |\tilde{\tau}_{(q_i, \Delta x_1)} - \tau^*_q|$. Suppose the first step provides the backward reachable set $\phi_1(x, t)$ while the second step gives us $\phi_2(x, t)$, we will find

$$
|\phi_2(x, \tilde{\tau}_{(q_i, \Delta x_2)}) - \phi(x, \tau^*_q)| \\
\leq |\phi_1(x, \tilde{\tau}_{(q_i, \Delta x_2)}) - \phi(x, \tau^*_q)| \\
\leq |\phi_1(x, \tilde{\tau}_{(q_i, \Delta x_1)}) - \phi(x, \tau^*_q)|
$$

(3.3)

As noted in Figure 3.5, the capture set on the dense grid is likely (but not guaranteed) to have a surface which has better accuracy than the capture set calculated on the sparse grid, as the progressing step size of the surface with dense grid will be much smaller than the one with sparse grid. Thus, it is necessary to decrease the estimate $\tilde{\tau}_{q_i}$ when beginning the computations on the dense grid. This paper assumes that the time for the capture set for a sparse and a dense grid are approximately equal, and thus will always deliberately underestimate $\tilde{\tau}_{q_i}$ when reusing an estimate from a sparse grid on a dense grid.

### 3.2.4 Parallelization

Several assumptions are put forth in order to understand decisions made later in the parallelization process.

**Assumption 3.2.1** *There is no inter-process communication between parallel computational processes, except to halt the process.*

This assumption permits the reuse of existing toolbox methods without inserting (time-consuming) logic to change computational parameters. Further, it permits direct use of the `parfor` command in MATLAB with no modifications or requirements for additional toolboxes.
Figure 3.5: The capture set $G_{(\tau^*_q, q_1)}$ for mode $q_1$ is calculated using a sparse grid (the solid line), and a dense grid (the dotted line). The dense grid calculation utilizes a smaller value of $\tau^*_q$ than that used in the sparse grid, in order to more tightly satisfy Property 3.2.1.

**Assumption 3.2.2** The computation process takes less time when the accuracy requirement is low (i.e., using flow estimation that utilizes only second order spatial and temporal approximations, as opposed to third or fifth order approximations).

As a datapoint, based on statistics from computational toolboxes, the third and fifth order approximations take approximately 3 times longer than the second order, and 17-18 times longer than a first order approximation[104].

**Assumption 3.2.3** The computation process takes less time when the grid size of the computation area is large (i.e., a sparse grid).

**Assumption 3.2.4** There exists at most one edge/transition entering each control mode, and at most one edge/transition exiting each control mode.

**Assumption 3.2.5** Once the system state enters a target, $T_i$, it can immediately transition into a new mode to reach the next target, $T_{i+1}$ as long as Property 3.2.2 and Property 3.2.3 are satisfied.
With these assumptions, it is possible to present an approach to parallelize the computation for an arbitrary number of control modes into which the system can transition from its initial state, \( N = ||Q|| \) called targets\(^1\), and an arbitrary number of parallel processors, \( p \), called nodes. Note that the initial state also has a target, \( T_0 \), which is used to verify Property 3.2.1 for the first control mode.

### 3.2.5 Single Target with Single Node

When \( ||Q|| = 1, p = 1 \), the algorithm is as discussed in Section 3.2.2. The algorithm is the following set of sequential computations:

![Diagram](image)

**Figure 3.6:** The computation algorithm for the computation of single target with single node.

\(^1\)\( N = ||Q|| \), because of Assumption 3.2.4.
This process can be repeated until the grid is as dense as the designer requires. Note that Property 3.2.3 is verified by the process of using the result of $C_G(\cdot)$ as the time value for computing $G$.

3.2.6 Single Target, Multiple Computing Nodes

When $||T|| = 1, p > 1$, it is possible to perform tasks from Section 3.2.5 in parallel, as long as information is not needed from the result of one computation in order to start the next. The result is a series of sequential tasks, each one of which is made up of parallel computations. At the end of each task, the results can be used to influence the parameters of future tasks; however, a parallel set of computations cannot influence one another (except to halt). One possible mapping, for $p = 2$ is:

In order to more densely express the process, parallel tasks are concisely written as follows:

$$G(\tau_{q_1}, q_1) || G(\tau_{q_1}, q_1)$$

Indicating that these two tasks execute concurrently, and that the next tasks are started once these complete.

The one node case shows that the total computation time needed depends on the initial estimate of $\tau$; however, the multi-node case can save time by computing the capture sets with several different values of $\tau$. If there are two node available, and the transition time estimation is $\tau_{q_1}$, then the first task could be $G(\tau_{q_1}, q_1) || G(\tau_{q_1}, q_1)$.

It is possible, using strategies such as these, to determine—in parallel—an accurate estimate of the lower bound, $\tau_{q_1}$ by ordering longer running tasks to halt once a computation determines that it has arrived at a lower bound estimate that was a result of underestimation.

If there are more than two nodes available for a single target, then the division of tasks can be further complicated. Suppose there are $N$ nodes ($N \geq 3$), the first computation will be

$$G(\tau_{q_1}, q_1) || \ldots || G(\tau_{q_1}, q_1) || G(\tau_{q_1}, q_1)$$

(3.4)
Figure 3.7: The computation algorithm for the computation of single target with multiple node.
After the first computation, if \( \frac{\tau_{q_1}}{2^{N-1}} < \tau_{q_1}^* \), then it is guaranteed that a lower bound \( \tau_{q_1}^* \) will be computed, permitting a computation of \( G(\tau_{q_1}^*, q_1) \) that satisfies Property 3.2.3.

If \( \frac{\tau_{q_1}}{2^{N-1}} \geq \tau_{q_1}^* \), then a new estimate \( \tilde{\tau}_{q_1}^{(1)} = \frac{\tau_{q_1}}{2^{N-1}} \) is set and the task is rerun. This process repeats until the lower bound is discovered. Each of these computations is performed using a sparse grid, to minimize loss of computation time.

3.2.7 Multiple Targets with Multi-node

When \( ||T|| > 1, p > 1 \), the following three rules are used to schedule tasks in sequence and parallel:

**Rule 1.** Concurrent parallel processes should require similar wall-clock time to complete;

**Rule 2.** Processes with low wall-clock time to complete should be finished first;

**Rule 3.** The schedule should reduce the overhead required by repeated computation.

The benefit of varying the grid size is its rapid result, albeit at the sacrifice of accuracy, while making full use of multi-node computation resources. Suppose we have \( p \) available computing nodes, and \( N \) targets to compute. Let \( \alpha_1 \) and \( \beta_1 \) represent the initial accuracy and grid spacing, while \( \alpha_2 \) and \( \beta_2 \) are the desired accuracy and grid spacing, where \( \alpha_1 < \alpha_2 \) and \( \beta_1 > \beta_2 \). The notation for the resulting capture and collision sets are augmented to include this information. With these constraints, the computation should generally follow the order below:

- **Group 1:** \( G(\tilde{\tau}_{q_1}, q_1, \alpha_1, \beta_1) || \cdots || G(\tilde{\tau}_{q_N}, q_N, \alpha_1, \beta_1) \);

- **Group 2:** \( G(\tilde{\tau}_{q_1}^{(1)} q_1, \alpha_2, \beta_2) || \cdots || G(\tilde{\tau}_{q_N}^{(1)}, q_N, \alpha_2, \beta_2) \);

- **Group 3:** \( G(\tilde{\tau}_{q_1}^*, q_1, \alpha_2, \beta_2) || \cdots || G(\tilde{\tau}_{q_N}^*, q_N, \alpha_2, \beta_2) \);

Group 1 is performed using the capture set calculation function, \( C_G(\cdot) \) for each state, and begins with the low accuracy, sparse grid estimates of \( \tilde{\tau}_{q_1}, \cdots, \tilde{\tau}_{q_1} \). The
results of the capture set calculation functions in the first task provide closer estimates to the lower bound \((\tilde{\tau}'_{q_1})\), which is revised to account for the sparseness of the grid using \(\tilde{\tau}'_{q_1} = \tilde{\tau}_{q_1} - \Delta \tau\).

Then, using \(\tilde{\tau}'_{q_1}, \ldots, \tilde{\tau}'_{q_N}\), the computations in Group 2 give sufficiently high accuracy calculations of the capture set, and the capture set calculation function returns the time lower bound with which each mode’s capture set was computed, namely \(\tau^*_{q_1}, \ldots, \tau^*_{q_N}\), with which the collision sets are computed in Group 3.

3.2.8 Variations of Group Scheduling

To arrange the groups, the number of nodes is also important. If \(p = N\), there is no need to arrange tasks, all the tasks in each group can be processed at the same time, according to the three tasks above.

If \(p < N\), then Group 1 may be divided into several sequential computations. At the \(m_{th}\) computation, if \(N = mp + n\), (and \(n < p\), at the \(m_{th}\) computation, there are still \(n\) tasks of Group 1 waiting for process. To take the advantage of multiple computing nodes, the first \(p - n\) tasks of Group 2 are calculated in the remaining \(p - n\) parallel slots, using the time lower bounds computed in Group 1, but the accuracy and grid density parameters for Group 2. Similarly, tasks may be loaded when Group 3 is processed.

If \(p > N\), there are more computational nodes than necessary computations in each group. Using an approach similar to the previous paragraph, tasks that compute \(G(\tau^*_q,q_1,\alpha_2,\beta_2), \ldots, G(\tau^*_{q_p-N},q_p-N,\alpha_2,\beta_2)\) are placed into the extra \(p - N\) parallel slots of Group 2.

3.3 Safety Verification Software: Design

3.3.1 Objective

The safety verification software is designed to model dynamic systems and generate verification software to check the satisfactory of system requirement and constraints.
3.3.2 Software Requirements

The verification software should provide an design interface for the user to model the dynamical system interaction. During the system modeling process, the software should also be capable of notifying the user system design constraint violations. Once the model is built, the designer will perform checks of the system’s verification requirements.

To check the system verification requirements, the software should be able to automatically generate code to perform the reachable sets computation. The user can specify the initial unsafe set of the system; based on that, the code will take advantage of LSM toolbox to speed up the computation of collision sets and capture sets. Thus designers will not have to know details of the LSM toolbox, they only need to consider system design, and computation result analysis.

Due to the structural and spatial complexity of reachable sets, the analysis of system behavior is difficult. The software should also provide a visualization tool for the designer to visualize capture sets and collision sets. This visualization tool should allow the user to make decisions to initiate an available transition.

The software should also be designed to be reusable for a family of dynamical systems. For example, a working model for cars can easily be reused to verify the safety of aircraft, a 1D model can be reused to verify 3D systems and so on.

3.3.3 Application Domain

This dissertation’s goal is to be capable of performing a safety check for at least 1D, 2D and 3D dynamical systems. The related capture sets and collision sets will be computed with time-dependent HJ-PDE through the LSM toolbox.
3.4 Domain Specific Modeling

3.4.1 Model-Based Software Synthesis

As the reachable set computation of each transition requires determination of Hamilton-Jacobi equation and configuration of parameters and initial condition, it is difficult to modify or reuse the system for a family of dynamic systems. In order to simplify the development of system verification software, domain-specific modeling (DSM) is implemented to model the multi-mode system and generate code for the computation.

DSM first requires a metamodel defined by the domain expert. The metamodel is the modeling paradigm for the model, which describes both semantic and syntactic behaviors of the model. Then the modeling environment can be generated through the interpretation of the metamodel. Finally a custom model interpreter semantically translates the model into the application domain.

Figure 3.8: Example multi-mode system where the thin circle represents the stationary state and the thick circle represents the transition state.

Figure 3.9 represents a model of the abstraction in Figure 3.8 with the domain specific tool. The guard conditions in Figure 3.8 can be inferred from the model.
structure in Figure 3.9. Using this modeling environment, executable parallel computation code is efficiently generated. Modification and reuse of the model are performed through the graphical interface, not through editing the generated code. Once a sufficient approximation for transition time $\tau_{q_{i+1}}^*$ is determined for each control mode, then the collision set $G(\tau_{q_{i+1}}^*, q_{i+1})$ may be calculated, permitting evaluation of Property 3.2.2 and Property 3.2.3, which is also called the 

**Definition 3.4.1** Let $C_s(T_{q_i}, T_{q_{i+1}}, \tau_{q_{i+1}})$ be called the **safety check function**. The safety check function checks whether the capture set $G(\tau_{q_{i+1}}^*, q_{i+1})$ and collision set $G(\tau_{q_{i+1}}^*, q_{i+1})$ each satisfy Property 3.2.2 and Property 3.2.3.

3.4.2 High Level Design

Due to the features of an embedded human system, the safety verification software will contain three main components: system abstraction, system parameter and computation. As in Fig. Figure 3.10, there are two processes for the interpreter to generate verification code, the first process is to parse the system abstraction and system parameter to represent the system with data objects, then the interpreter will generate verification code by hydrating the computation model with data from
The system abstraction component shall provide user an interface that can model the system. As the Embedded Human System is decomposed into steady states and transition modes connecting steady states, special data structures shall be designed by the DSM designer to represent steady states and transition modes. For example, the data structure for transition mode shall contain controller logic and controller gains, the data structure for steady state shall contain state variables.

The system parameter component shall allow the user to add, modify and delete system parameters. For example, the verification of car dynamics system will at least require maximum and minimum velocity and turning rate of the specific car model. Once we want to reuse the verification code to a similar domain, we may only need to modify these system parameters.

The computation component stores the configuration information of reachable set computation. The dimensional information, Hamiltonian function and partial function shall be put into this component, to generate computation scripts for each transition mode.

![Diagram of System Components](image)

Figure 3.10: Interaction of three main components for the safety verification software.
3.4.3 Meta-model

The meta-model of domain specific modeling is supposed to define entities, attributes and relationship between entities in a simple and precise way. After the interpretation of meta-model, the paradigm for the modeling environment can be generated, thus the modeling environment can effectively constrain the model builder to ensure that the syntax and semantics of the model are correct, and at the same time the paradigm should also provide enough freedom and flexibility to the end-user; Thus the paradigm can be applied to a family of systems. The meta-model is defined by domain experts, and the end-user only changes the model under the constraints of the paradigm.

Using the domain-specific modeling tool GME, the design space of the model can be decomposed into several aspects with different types, thus different structures and behaviors can be clearly modeled. In the design of the meta-model, the domain expert shall consider how to represent the semantic information with the model built under the paradigm generated from meta-model, and how to extract the information from the model.

In GME, the meta-modeling interface allows domain experts to create graphical entity-relationship diagrams. As in Figure 2.4, the domain expert can drag different entities from the Part Browser in the left, and set different relationships between these entities. To simplify the modeling environment, meta-model of UAV refueling system is decomposed into four models: State Transition Model (Figure 3.15), Reachable Set Setup Model (Figure 3.16), Behavior Model (Figure 3.11) and Controller Model (Figure 3.17).

In Figure 3.11, the behavior model class diagram of the meta-model is shown. The ModelProxy in the class diagram can be considered as Model References, and there are three ModelProxies and one Atom object in this model. This class diagram allows the StateTransitionDiagram Model to contain Constant objects. The behavior model attribute is used to set a couple of fields for three Model References and the Atom. In Figure 3.12, several different Models contain the same Field Attributes
Figure 3.11: The Behavior Model class diagram of meta-model.

Figure 3.12: The Behavior Model attribute of meta-model.
Figure 3.13: The Behavior Model visualization of meta-model.

Figure 3.14: The Behavior Model visualization configuration mode of meta-model.
such as “HamFunction” and “PartialFunction”. The Behavior Model Visualization sets the object that can be seen by modeler, in Figure 3.13 the StateTransitionDiagram Model is set to contain Behavior Aspect, and the objects available to modeler can be set in configuration mode as in Figure 3.14 (the greyed objects will be hide from the modeler).

In Figure 3.15, the state transition class diagram of the meta-model is displayed. There are three Models and two Connections. Model StateTransitionDiagram contains two other Models: State Model and TransitionMode Model. There are two Connections between State Model and TransitionMode Model: State2Transition Connection and ContinueTransition2State Connection. With these entities and relationships in meta-model, we can add several States and TransitionModes objects into the StateTransitionDiagram object, and directional connections are allowed between State object and TransitionMode object.

The reachable set modeling component of meta-model is as in Figure 3.16, three Model objects are added to the StateTransitionDiagram Model. TargetSet Model represents backward reachable set which has SafeOrUnsafe option for the specification of reachable set type. InvSet Model represents invariant set computation. The StateVariable Atom represents the dimensional information of the computation, and GenericTargetShape Atom allows user to specify the shape of initial target set.

The controller settings for model are defined in Controller Model as in Fig-
Figure 3.16: The ReachSetSetup Model class diagram of meta-model.

Figure 3.17: The Controller Model class diagram of meta-model.
ure 3.17, where the Controller Model is allowed to have different types of components such as Gain Atoms, Tolerance Atoms and Constant Atoms. As the Abstract-Controller object shall be inherited by both Model and Reference, it is defined as FCO(First Class Object). In this class diagram, as the State Model and Transition-Mode Model both are subtypes of AbstractMode Model, they are allowed to contain AbstractController FCOs, Constant Atoms, Tolerance Atoms and Gain Atoms as the AbstractMode Model.

3.4.4 Constraints

The constraint feature of meta-model can be implemented to avoid some error made by the modeler early in the model building process. In meta-model, constraints can be added to the model by attach constraint object to the related object as in Figure 3.18. Descriptions and Equations using OCL can be added to the constraint as in Figure 3.19, as the TransitionMode Model only allows one input and one output, “self.bagConnectedFCOs(State2Transition)->size < 2” is used to describe this condition. In the modeling environment, once we try to make

Figure 3.18: Constraints can be added to the meta-model.

a second input connection or output connection to the TransitionMode Model, a
Figure 3.19: The constraint can be described by OCL (Object Constraint Language).

Figure 3.20: A warning window will pop out once the constraint is violated in model design.

3.4.5 Model Design

Once the meta-model is available, MetaGME Interpreter can be used to translate the meta-model into domain-specific modeling environment. The user interface of domain-specific modeling environment is as in Figure 3.21, there are four modeling aspects: StateTransition Aspect, Behavior Aspect, ReachsetSetup Aspect and ControllerSetting Aspect. In each aspect only a small group of objects will be modified and the appearance is thus more concise.
To build the system for safety check, the modeler should first create a StateTransitionDiagram Model at the RootFolder. Once we open the StateTransitionDiagram Model, we can find the Part Browser at the top left corner has three tabs: State-Transition, Behavior and ReachSetSetup.

There are two types of objects under StateTransition tab: State and TransitionMode. An abstracted state transition system can be built as in Figure 3.21 by dragging objects from the Part Browser and making directional connections among them. Inside State Model or TransitionMode Model, the lower level structures of these models can be constructed under ControllerSetting Aspect as in Figure 3.22, for example, Constant objects can be added into the TransitionMode to represent target location information. In Figure 3.22, a moving controller is also added to the TransitionMode. Inside the moving controller in Figure 3.23, the internal parameters such as gain, tolerance and target locations of the controller can be added.

![Image](image-url)

Figure 3.21: The state transition aspect of modeling environment.

In behavior aspect the system specifications can be added, Figure 3.24 shows a system model with six Constant objects representing system specifications such as
tanker velocity range, UAV velocity range and turning rate range.

In reachable set aspect in Figure 3.25, the state space and desired result reachable sets can be set. The StateSpace object is a basic component in this aspect as it contains dimensional information, the desired result reachable sets includes capture set, collision set and invariant set. Capture set and collision set can be represented by TargetSet objects with its SafeOrUnsafe option, invariant set can be represented by InvSet object.

3.4.6 Code Generation

The final product in the application domain is generated with the interpretation of the model. The interpreter applies computational transformation to the model to map the model into the application domain. The interpreter shall be able to extract system information and translate the information into objective code or script. The structures of interpreter for general dynamic system safety verification are as bellow,
two classes **SafetyCheck** and **ReachSet** are implemented to store the system data and interpret that into MATLAB code.

The UML diagram of **ReachSet** class is as in Figure 3.26. Each **ReachSet** object represents the computation process for one reachable set (either Capture Set or Collision Set). All the required information for one reachable set computation are stored in its data fields, such as the boolean value **isSafe** identifies the type of reachable set (true for capture set and false for collision set), the PDE is represented by string **partialFunc** and the Hamiltonian function is represented by string **hamFunc**. The listed methods in Figure 3.26 are implemented to generate the MATLAB script for reachable set computation, different methods set different parts of the computation script. For example, the **setBeginning()** method generates the beginning part of the script, which checks the number of input arguments and set default values if there are not enough input arguments, the **setMain()** method generates the main loop of reachable set computation, different scripts will
Figure 3.24: The behavior aspect of modeling environment.

Figure 3.25: The Reachable Set aspect of modeling environment.
Figure 3.26: The UML diagram for ReachSet class, where only important attributes and methods are listed.
be generated base on the type of \texttt{i}sSafe, the main loop for capture set computation will check whether the initial state is included by the capture set that grows to the set time (if not, more transition time will be added to expand the capture set).

![RouteNode](image1)

**Figure 3.27:** The UML diagram for RouteNode class.

![SafetyCheck](image2)

**Figure 3.28:** The UML diagram for SafetyCheck class, where only important attributes and methods are listed.

The class diagram for \texttt{RouteNode} and \texttt{SafetyCheck} classes are as in Figure 3.27 and Figure 3.28. The \texttt{RouteNode} class is used to store the name for transition mode, source state name and destination state name. \texttt{SafetyCheck} class stores transition route information inside a vector(\texttt{routeTable}) of RouteNode pointers. The \texttt{SafetyCheck} class is designed to represent a whole transition
state diagram. Each transition mode in GME is visited by visitTransitionMode method to get the constants and controller information, each state will also be visited to obtain the initial state and final state information, these information will be used to construct ReachSet objects and depends on the type of the objects, they will be stored in reachsets vector or escapesets vector. Once the reachsets vector or escapesets vector are available, generateCode methods will be called to generate computation codes, the input argument count of this method defines the step of the computation codes. The generateAll method generates code for computation management, for example, if the system in GME is configured to have four core computation resource, the generateAll method will organize the script to take advantage of the four core computation capability, if the system has a huge number of transition modes, the computation script will always have four computation process running in parallel.

The generateParameterInit, generateRouteTableInit and generateManeuver methods generate scripts for parameter initialization, route table initialization and transition trajectory computation. generateParameterInit method generates a MATLAB script containing system behavior information as in listing 3.1. generateRouteTableInit generates a MATLAB script containing transition route information, steady state list and a name list for saved data. generateManeuver generates a switch function to select maneuvers for transition process or escape process and related trajectory computation functions as in listing 3.3 and 3.4

Listing 3.1: parameterInit.m

```matlab
1 % Here is the parameter initialization list
2 k1 = 3;
3 k2 = 1;
4 vmax_tanker = 80;
5 vmax_uav = 120;
6 vmin_tanker = 60;
7 vmin_uav = 40;
8 yawmax_uav = pi/3;
9 yawmin_uav = -pi/3;
```

Listing 3.2: routeTableInit.m
Here is the route table initialization list

\begin{verbatim}
gInfo1 = 'simple.m1.mat';
gInfo2 = 'simple.m2.mat';
gInfo3 = 'simple.m3.mat';

nameTable = {'rs_TransitionMode2_s2_to_s3';
             'rs_TransitionMode3_s3_to_s4';
             'rs_TransitionMode1_s1_to_s2';
             'rs_TransitionMode4_s2_to_s4';
             'us_TransitionMode2_s2_to_s3';
             'us_TransitionMode3_s3_to_s4';
             'us_TransitionMode1_s1_to_s2';
             'us_TransitionMode4_s2_to_s4'};

waypoints = [30 5 0; 25 33 0; 10 40 0; 5 30 0];

routeTable = [2 3; 3 4; 1 2; 2 4];
\end{verbatim}

Listing 3.3: maneuver2ode.m

\begin{verbatim}
switch maneuver
  case 1
    [t,X] = ode23(@ode_transition0,[0,rsTime{1}{end}],[locx1_prev;locx2_prev;0]);
  case 2
    [t,X] = ode23(@ode_transition1,[0,rsTime{1}{end}],[locx1_prev;locx2_prev;0]);
  case 3
    [t,X] = ode23(@ode_transition2,[0,rsTime{1}{end}],[locx1_prev;locx2_prev;0]);
  case 4
    [t,X] = ode23(@ode_transition3,[0,rsTime{1}{end}],[locx1_prev;locx2_prev;0]);
  otherwise
    sprintf('There is something wrong with ode functions. ');
end
\end{verbatim}

Listing 3.4: escape2ode.m

\begin{verbatim}
switch mod(escape,5)
  case 2
\end{verbatim}
As the MATLAB code for system verification and visualization can be separated into two groups: model-dependent scripts and model-independent scripts. The model-dependent scripts include scripts of reachable sets computation, parameter initialization script, system transition table script and trajectory computation functions. As the model-dependent scripts are directly related to specific system, they must be generated by the interpreter once a new model needs verification. The model-independent scripts include a couple of functions that mainly interact with the model-dependent scripts and data from reachable sets computation to visualize different modes, such as the result data loading functions, transition visualization functions, steady state visualization functions and so on. These functions can be generalized to visualize different dynamics system or different transition protocols. In order to reduce the development time, these model-independent functions are manually created, and they can be used with any model-dependent scripts and data to visualize different processes.

The data loading function `loadReachsets` is as in listing B.2. The input arguments `i` defines the row number of the required reachable set data, after running the `routeTableInit`, the data loading function can obtain the name of capture set data and collision set data from `nameTable` in listing 3.2, the initial state and final state data are also provided in `waypoints` and `routeTable`.

The actual trajectory for each transition in system visualization is computed with ODE functions as in listing 3.3 and 3.4. These ODE function selection scripts and
ODE functions are all generated by `generateManeuver` method of `SafetyCheck` class.

To demonstrate the capabilities of the Safety Check Tool as in Figure 3.34, a couple of use cases are tested in next chapter, such as verification of a single platform Figure 3.29, reusing a working model for other controllers Figure 3.30 or other platforms Figure 3.31, protocol restoration Figure 3.32 and reusing model for multiple platforms Figure 3.33. The relationship between use, Safety Check Tool and Visualization Tool in these use cases are as in Figure 3.35, Figure 3.36, Figure 3.37, Figure 3.38 and Figure 3.39. The activity diagrams for single platform verification and multiple platforms or controllers verification(with model reusing) are as in Figure 3.40 and Figure 3.41.
**Use Case:** Verify the safety of state transition for a single platform

**Summary:** Target shall finish transition for a sequence of states and avoid unsafe state during the process.

**Actors:** Model designer

**Preconditions:** The dynamic equations, Hamiltonian equations and other related system behaviors shall be provided.

**Description:** The model designer shall build the verification system following the entities and relationships defined in meta-model. Once the model is built, the interpreter is called to generate scripts for system requirements verification. The user shall initiate the generated scripts to start the computation of reachable sets, and verify the system with visualized computation results when the computation is completed.

**Exceptions:**

*Constraints Violation:* If the model designer violates certain system design constraints, a warning message will pop out to stop it. For example, when user tries to make undefined connections between two nodes, a warning message will appear.

*Interpreter Fails:* If the model contains faults, the interpreter may fail to generate scripts. For example, if the model is missing certain necessary parameter, an error message will appear to notify user the fault.

*Computation Fails:* If the scripts are generated but the model is not carefully configured, it is still possible that the computation may not run correctly or it may run but not generate required results. In this case, the user shall modify the model to correct the configuration and regenerate the computation scripts.

*Verification Fails:* Once the computation results are available, the visualization will display the reachable set as well as the moving trajectory of the target. If the visualization shows some unwanted states, the user shall modify the model base on the visualization.

**Postconditions:** The model is available for future reuse.

Figure 3.29: **Use Case 1 Description.** The use case description for verification of a single platform.
Use Case: Reuse the working model for other controller verification

Summary: Target shall finish transition for a sequence of states and avoid unsafe state during the process.

Actors: Model designer

Preconditions: A working model in the same or similar domain shall be provided.

Description: Reuse the available model by modifying controller behaviors. Once the updated model is built, the interpreter is called to generate scripts for system requirements verification. The user shall initiate the generated scripts to start the computation of reachable sets, and verify the system with visualized computation results when the computation is completed.

Exceptions:

Computation Fails: If the scripts are generated but the model is not carefully configured, it is still possible that the computation may not run correctly or it may run but not generate required results. In this case, the user shall modify the model to correct the configuration and regenerate the computation scripts.

Verification Fails: Once the computation results are available, the visualization will display the reachable set as well as the moving trajectory of the target. If the visualization shows some unwanted states, the user shall modify the model base on the visualization.

Postconditions: The model is available for future reuse.

Figure 3.30: Use Case 2 Description. The use case description for reusing the working model for other controllers.
Use Case: Reuse the working model for other platform verification

Summary: Target shall finish transition for a sequence of states and avoid unsafe state during the process.

Actors: Model designer

Preconditions: A working model in the same or similar domain shall be provided.

Description: Reuse the available model by modifying system behaviors. Once the updated model is built, the interpreter is called to generate scripts for system requirements verification. The user shall initiate the generated scripts to start the computation of reachable sets, and verify the system with visualized computation results when the computation is completed.

Exceptions:

Computation Fails: If the scripts are generated but the model is not carefully configured, it is still possible that the computation may not run correctly or it may run but not generate required results. In this case, the user shall modify the model to correct the configuration and regenerate the computation scripts.

Verification Fails: Once the computation results are available, the visualization will display the reachable set as well as the moving trajectory of the target. If the visualization shows some unwanted states, the user shall modify the model base on the visualization.

Postconditions: The model is available for future reuse.

Figure 3.31: Use Case 3 Description. The use case description for reusing the working model for other platform verification.
Use Case: Redesign the transition protocol base on the previous verification results

Summary: Target shall finish transition for a sequence of states and avoid unsafe state during the process.

Actors: Model designer

Preconditions: The original model and related verification results (fail or partially fail) for this platform shall be provided.

Description: Reuse the model by redesign the failed transition protocol (For example, insert middle steady states to failed transitions or modify the controllers for failed transitions). Once the updated model is built, the interpreter is called to generate scripts for system requirements verification. The user shall initiate the generated scripts to start the computation of reachable sets, and verify the system with visualized computation results when the computation is completed.

Exceptions:
Computation Fails: If the scripts are generated but the model is not carefully configured, it is still possible that the computation may not run correctly or it may run but not generate required results. In this case, the user shall modify the model to correct the configuration and regenerate the computation scripts.

Verification Fails: Once the computation results are available, the visualization will display the reachable set as well as the moving trajectory of the target. If the visualization shows some unwanted states, the user shall modify the model base on the visualization.

Postconditions: The model is available for future reuse.

Figure 3.32: Use Case 4 Description. The use case description for reusing original model to restore transition protocols.
Use Case: Verify shared operation behavior for multiple platforms

Summary: Multiple platforms shall all finish transition for a sequence of states and avoid unsafe state during the process, the shared capture sets or collision sets for all these platforms shall have the desired behaviors.

Actors: Model designer

Preconditions: The dynamic equations, Hamiltonian equations and other related system behaviors shall be provided for the first platform, other platforms can reuse the model of the first platform.

Description: The model designer shall build the verification system following the entities and relationships defined in meta-model for the first platform and reuse the model for other platforms. Once the model is built, the interpreter is called to generate scripts for system requirements verification. The user shall initiate the generated scripts to start the computation of reachable sets, and verify the system with visualized computation results when the computation is completed.

Exceptions:

Constraints Violation: If the model designer violates certain system design constraints, a warning message will pop out to stop it. For example, when user tries to make undefined connections between two nodes, a warning message will appear.

Interpreter Fails: If the model contains faults, the interpreter may fail to generate scripts. For example, if the model is missing certain necessary parameter, an error message will appear to notify user the fault.

Computation Fails: If the scripts are generated but the model is not carefully configured, it is still possible that the computation may not run correctly or it may run but not generate required results. In this case, the user shall modify the model to correct the configuration and regenerate the computation scripts.
**Verification Fails:** Once the computation results are available, the visualization will display the reachable set as well as the moving trajectory of the target. If the visualization shows some unwanted states, the user shall modify the model base on the visualization.

**Shared Behavior Fails:** Suppose all platforms pass verification, but the shared capture set (intersection of individual capture sets) or collision set (union of individual collision sets) does not satisfy desired behaviors. The platform that causes the undesired behavior shall be modified.

**Postconditions:** The model is available for future reuse.

---

**Figure 3.33: Use Case 5 Description.** The use case description for verification of shared operation behavior for multiple platforms.

**Figure 3.34:** The use case diagram for safety check tool.
Figure 3.35: The sequence diagram of code generation process.

Figure 3.36: The sequence diagram of computation and result visualization process.

Figure 3.37: The sequence diagram of reusing a working model.
Figure 3.38: The sequence diagram of reusing a failed or partially failed model.

Figure 3.39: The sequence diagram of reusing model for different platforms.
Figure 3.40: The activity diagram for single platform verification.
Figure 3.41: The activity diagram for multiple platform and multiple controller verification.
CHAPTER 4

Results

4.1 Parallel Computation

All presented simulation results are the mean value of 10 executions of the overall algorithm using a high-performance computing cluster made up of 8-core SGI Altix ICE machines. In order to provide relevant results to high-end desktop machines that feature multiple cores, but may have issues with shared memory when all cores are active, results are presented using both a number of cores on the same node, and a number of nodes (each of which is allocated one core).

The simulation of cooperative planar aircraft collision avoidance between two aerial vehicles is used to test the above algorithm. This example utilizes the Hamiltonian and dynamics given in [83]. One aircraft is a manned vehicle, which aims at straight and level flight. The other vehicle is a UAV which aims to move in sequence to several positions relative to the manned in a prescribed order. Proportional controllers are used in each of the various modes to move to setpoints that prescribe the waypoint motion with respect to the relative coordinate system described in [83].

The MATLAB \texttt{parfor} function provides the capability of parallel processing, and as the previous assumption states, there is no inter-process communication between the processes of multi-node.

4.1.1 Single Target

The initial location of the UAV is set to be (0, 0), and the target to be (10, 0), while the lower bound of transition time is assumed to be between 1s and 4s. The results of execution with 1-node, 2-nodes and 4-nodes are in Figure 4.1, and the results of 1-core, 2-cores and 4-cores are in Figure 4.2. In Figure 4.1, Figure 4.2, the bold numbers are the time needed to finish each step, which is equivalent to the most
Figure 4.1: Single target, with multiple available computing nodes.

Figure 4.2: Single target, single computer with multiple cores.
time-consuming process in each task. The histogram of single target results is shown in Figure 4.3.

From the histogram, it is clear that the use of parallel processing, even for a single target, saves significant wall clock and computation time. Both capture and collision sets are computed, and the estimated time lower bound for completing each control mode is unknown prior to computation. Figure 4.3 is a graphical comparison of the difference of performance between single and multi-node/multi-core results. Note that for 1-node/1-core, the 1-node performs significantly better than 1-core, because MATLAB will utilize implicit multiprocessing through calls to a multithreaded implementation of the basic linear algebra subprograms (BLAS) library on a second core (if it exists).

4.1.2 Multi-target

The multi-target case has two accuracy levels, with a low level that uses a sparser computation grid while the high accuracy level uses a denser grid, the lower bounds of transition time are all assumed to be between 1s and 4s. The results of multi-target parallelization are shown in Figure 4.4, Figure 4.5, and Figure 4.6. For brevity, the tabular results for the 2, 4, 8-node cases are shown only in the figure. The computation time is shown as a histogram in Figure 4.7.
Figure 4.4: Results of 4 targets with 2 nodes.

Figure 4.5: Results of 4 targets with 4 nodes.

Figure 4.7 shows the total computation time of the multi-node case is not reduced as drastically as in the multi-core case. That is, in Figure 4.3, the 2-core case cuts nearly in half the total time of 1-core case, and 4-core uses approximately half of the total time of 2-core case.

The above results show that, with the presented time estimation methods and task arrangement, multi-node and multi-core machines efficiently reduce the computation time required to compute the capture and collision sets, in a way that verifies the presented properties of the controller.

4.2 Examples

To demonstrate the capabilities of the safety verification tool, several examples in different dimensions are verified below. The functionalities verified for each example is as in Table 4.1:
Figure 4.6: Results of 4 targets with 8 nodes.

Figure 4.7: Computation time of 4 targets when there are multiple cores and nodes. (see Figure 4.4, Figure 4.5 and Figure 4.6)
Use Case | Example | 1D Example | 2D Example | 3D Example 1 | 3D Example 2
--- | --- | --- | --- | --- | ---
verify a single platform | √ | √ | √ | √ |
reuse the working model for other controllers | √ |
reuse the working model for other platforms | √ | √ | √ | √ |
reuse original model to restore transition protocols | | | | √ |
verify shared operation behavior for multiple platforms | √ | √ | √ | √ |

Table 4.1: Use Cases tested for different examples.

4.2.1 1D Example

Suppose we have a 1D person model, the person is only allowed to walk back or forth in a straight line. The person can choose the walking velocity as he/she wants within the velocity range.

The dynamics of the 1D person model is \( \frac{dx_1}{dt} = u \), where \( x_1 \) is the horizontal distance and \( u \) is the velocity. The Hamiltonian of the system is \( H(x, p) = p_1 u \).

Two person models with different velocity ranges are simulated, they both use a P controller with saturation for the velocity input \( \max(\min(k_1(x_{1t} - x_1), v_{max}), v_{min}) \).

Person 1 has velocity range \( v_{min} = -0.5 m/s \) and \( v_{max} = 2 m/s \), person 2 has velocity range \( v_{min} = -1 m/s \) and \( v_{max} = 1.5 m/s \). The system is as in Figure 3.9, where there are four states (with horizontal distance 30, 15, 5 and 25) and four transitions.

The objective of the system verification is to observe the behaviors of each transition with reachable sets computation using Domain Specific Modeling. The verification results of two person models are shown in Figure 4.9, Figure 4.10 and Figure 4.11.

In the verification, in order to make the visualization of reachable sets easier, an extra dimension is added to the model and its dynamic equation \( \frac{dx_2}{dt} = 0 \) is used to compute the capture sets. In Figure 4.9, Figure 4.10 and Figure 4.11, the capture sets describes the minimum sets of initial conditions that could reach the target state.
and cover the initial state. As person 1 and person 2 have different specifications, the result capture sets have different shapes. For example, the reachable set of person 1 is larger than person 2, this is because the initial state shall move in negative direction to reach the target, as the maximum negative velocity of person 1 is lower than person 2, person 1 has larger minimum reach time than person 2. As person 1 also has higher positive velocity than person 2, person 1’s capture set covers more area than person 2 at the left side. Base on these results, the same system can be verified and applied to different models. It is common that one system design that works for one model may violate the design specification for another model. For instance, the transition times in Figure 4.11 are about the same (24s) for both models, if the initial state is changed to [40, 0] and the transition time should be less than 24s, the design will still work for person 2, however, the design will no longer work for person 1, as [40, 0] is out of the person 1’s capture set.
Figure 4.8: GME model for 1D example. Using the same model as in Figure 3.9, the system can be built with modifications to the States, TransitionMode and system behavior descriptions. The location information can be added to the State block, different controllers and their parameters shall be added to each TransitionMode and output parameters such as available number of cores for computation and output folder, system dynamic equations, Hamiltonian and derivative equations are also provided to the system behavior descriptions at the bottom right corner of the model builder.
Figure 4.9: Minimum capture sets of transition starts from State 1 for two 1D person models. As the maximum backward velocity of Person 1(-0.5m/s) is lower than Person 2(-1m/s), the minimum reach time of Person 1(33.98sec) is longer than Person 2(23.72sec). Due to the higher maximum forward velocity(2m/s for Person 1 and 1.5m/s for Person 2) and longer minimum reach time, the minimum capture set of Person 1 contains more area than Person 2.
4.2.2 2D Example

The 2D person model can choose the walking velocity and heading angle, thus the walking trajectory has two dimensions. The dynamics is as following:

\[
\frac{dx_1}{dt} = u_1 \cos(u_2) \quad (4.1)
\]
\[
\frac{dx_2}{dt} = u_1 \sin(u_2) \quad (4.2)
\]
\[
(4.3)
\]

There are two control inputs, \( u_1 \) is velocity input and \( u_2 \) is the heading input. The 2D dynamics can be effectively extended from the 1D dynamics system by modifying certain fields in the graphical interface. For example, the four states of the system are modified to (42, 15), (25, 40), (10, 38) and (5, 25). The controllers are also modified as following:

\[
u_1 = \max(\min(k_1(x_{1t} - x_1), v_{max}), v_{min}) \quad (4.4)
\]
\[
u_2 = \tan^{-1}\left(\frac{x_{2t} - x_2}{x_{1t} - x_1}\right) \quad (4.5)
\]

Three person models are implemented in this system, person 1 and person 2 are using the above controller, while person 3 is using a different control logic. The controller for person 3 is as following:

\[
u_1 = \max(\min(k_1(x_{1t} - x_1), v_{max}), v_{min}) \quad (4.6)
\]

\[
u_2 = \begin{cases} 
\pi & \text{if } x_1 > x_{1t} \\
0 & \text{if } x_1 \leq x_{1t} \\
\pi/2 & \text{if } x_1 = x_{1t}, \ x_2 \leq x_{2t} \\
-\pi/2 & \text{if } x_1 = x_{1t}, \ x_2 > x_{2t}
\end{cases} \quad (4.7)
\]

The control logic for person 1 and person 2 are using P controller for velocity input and make the heading input the direction from current location towards the target. The heading input of person 3 let the person 3 first move horizontally towards the target, then move vertically to reach the target.
Figure 4.10: Minimum capture sets of transition starts from State 2 for two 1D person models. The top figure shows the minimum capture set for Target 1, the bottom figure shows the minimum capture set for Target 2. Target 1 locates at left side of initial state and Target 2 locates at the right side of initial state. Due to the velocity differences in forward and backward directions, Person 1(22.50sec) has longer minimum reach time for Target 1 than Person 2(17.64sec), Person 2(17.92sec) has similar minimum reach time for Target 2 than Person 1(17.93sec). Behaviors for different decisions can be observed with minimum reach time and the shape of capture sets.
Figure 4.11: Minimum capture sets of transition starts from State 3 for two 1D person models. The shape of capture sets is similar as the shapes in transition from State 2 to Target 2 as in Figure 4.10, as these transitions implement the same controller and they both move forward to reach the target.
In order to verify different behaviors of these models, person 1 and person 3 have the same velocity range $v_{\text{min}} = 1\text{m/s}$ and $v_{\text{max}} = 2\text{m/s}$ as they implement different controllers, person 2 has $v_{\text{min}} = 0.5\text{m/s}$ and $v_{\text{max}} = 1\text{m/s}$ as it uses the same control logic as person 1.

To use the safety check tool, the Hamiltonian of this system is set as following:

$$H(x, p) = p_1 u_1 \cos(u_2) + p_2 u_1 \sin(u_2)$$  \hspace{1cm} (4.8)

After system verification, the reachable sets of three person models are shown in Figure 4.13, Figure 4.14 and Figure 4.15. From these figures, the capture sets of Person 1 and Person 2 have small differences due to the difference in input ranges while the capture set of person 3 have completely different shape due to the different control logic.
Figure 4.13: Minimum capture sets of transition starts from State 1 for three 2D person models. The minimum capture sets of Person 1 and Person 2 have similar shapes, as they both implement the same control logic, due to their differences in velocity range, they have different minimum reach time (Person 1 has reach time 20.57 sec, Person 2 has reach time 34.50 sec). Person 3 has a completely different minimum capture set shape because of its special control logic, and Person 3 also has longer minimum reach time (42.36 sec). As both control logics are symmetric about x axis and y axis, we can observe that the capture sets are also symmetric about x axis and y axis.
4.2.3 3D Example 1

Suppose there is a 3D car model, there are three components in its state space: horizontal distance, vertical distance and heading angle. The dynamics of the system is as following:

\begin{align}
\frac{dx_1}{dt} &= u_1 \cos(x_3) \\
\frac{dx_2}{dt} &= u_1 \sin(x_3) \\
\frac{dx_3}{dt} &= u_2
\end{align}

There are two inputs for the car, \( u_1 \) is the speed of the car and \( u_2 \) is the turning rate of the car. Here two different cars are modeled, car 1 has \([5 \text{m/s} \ 30 \text{m/s}]\) as its speed range and \([-\pi/2 \text{rad/s} \ \pi/2 \text{rad/s}]\) as its turning rate range. Car 2 has speed range \([10 \text{m/s} \ 20 \text{m/s}]\) and turning rate range \([-\pi/3 \text{rad/s} \ \pi/3 \text{rad/s}]\).

The control logics for both car models are as following (suppose \( x_3 \) and \( x_{3t} \) are in range \([-\pi \ \pi])):

\begin{align}
u_1 &= \max(\min(k_1(x_{1t} - x_1), v_{\text{max}}), v_{\text{min}}) \\
u_2 &= \begin{cases} k_2(x_{3t} - x_3) & \text{if } |x_{3t} - x_3| \leq \pi \\ -\text{sign}(x_{3t} - x_3)k_2(2\pi - |x_{3t} - x_3|) & \text{if } |x_{3t} - x_3| > \pi \end{cases}
\end{align}

And the Hamiltonian function for reachable sets computations are as following:

\[ H(x, p) = p_1u_1\cos(x_3) + p_2u_1\sin(x_3) + p_3u_2 \]

The verification results for all four transitions are shown in Figure 4.17, Figure 4.18 and Figure 4.19, as the actual capture sets are 3D shapes, to make the visualization and comparison easy, these figures are all taking one slice at \( x_3 = 0 \) of the capture sets to compare their behaviors. The abstracted state transition model
Figure 4.14: Minimum capture sets of transition starts from State 2 for three 2D person models. As this transition model is from reusing the previous example, there are also two target states for State 2. As target 1 is closer to initial state than target 2, the minimum reach time for target 1 is generally longer than for target 2.
Figure 4.15: Minimum capture sets of transition starts from State 3 for three 2D person models. The shapes of minimum capture sets are different from previous transitions starting from State 1 and State 2, as the minimum reach time for this transition (14.07sec for Person 1, 25.71sec for Person 2 and 28.50sec for Person 3) is shorter than transition from State 1 and transition from State 2 to target 1, but longer than transition from State 2 to target 2.
of the system is as in Figure 3.9. The four states are [5, 5, 0], [20, 15, 0], [30, 10, 0] and [45, 10, 0].

From the capture sets data, as car 1 has higher maximum velocity and turning rate, the minimum capture time for car 1 is a bit smaller than car 2, and the shape of capture set of car 1 is larger than car 2.
Figure 4.17: Minimum capture set slices at $x_3 = 0$ of transition starts from State 1 for three 3D car models. The minimum reach time for Car 1 and Car 2 are 1.71sec and 1.55sec.
4.2.4 3D Example 2

The UAV refueling example contains UAV and tanker, the purpose of the verification is to avoid collision during the whole refueling process and make the refueling efficient. As both UAV and tanker are moving, to simplify the analysis, tanker reference frame is used to describe the dynamic behaviors. The system state $x$ is the relative location of tanker with respect to the UAV. The dynamic equations are as following:

\[
\frac{dx_1}{dt} = -u_1 + v_t \cos(x_3) + u_2 x_2 \tag{4.15}
\]
\[
\frac{dx_2}{dt} = v_t \sin(x_3) - u_2 x_1 \tag{4.16}
\]
\[
\frac{dx_3}{dt} = -u_2 \tag{4.17}
\]

where $v_t$ is the velocity of tanker, which is set to $80\,m/s$ in the verification. Input $u_1$ is the UAV velocity and input $u_2$ is the UAV turning rate. P controllers are used for control inputs as following:

\[
u_1 = \max(\min(k_1(x_{1t} - x_1), v_{max}), v_{min}) \tag{4.18}\]
\[
u_2 = \max(\min(k_2(x_{2t} - x_2), v_{max}), v_{min}) \tag{4.19}\]

And the Hamiltonian of the system is as following:

\[H(x, p) = p_1(-u_1 + v_t \cos(x_3) + u_2 x_2) + p_2(v_t \sin(x_3) - u_2 x_1) + p_3(-u_2) \tag{4.20}\]

The abstracted state transition model of the system is as in Figure 3.9. The four states are located at $[30, 5, 0]$, $[25, 33, 0]$, $[10, 40, 0]$ and $[5, 30, 0]$. There is a unsafe zone that circles around the tanker which is called minimum separation set, collision may happen if UAV enters this set. In this example, the set contains any initial states that will enter the minimum separation set is defined as collision set.

The above examples are all taking advantage of the automatically generated code from domain specific interpreter. As the interpreter has integrated the parallel
Figure 4.18: Minimum capture set slices at $x_3 = 0$ of transition starts from State 2 for three 3D car models. The minimum reach times for Car 1 and Car 2 in the top figure are both 1sec. The minimum reach times for Car 1 and Car 2 in the bottom figure are 1sec and 1.21sec. As the initial estimate of minimum reach time for all these transitions is 1sec, it can be inferred that the actual minimum reach times for both cars in top figure and Car 1 in bottom figure are all less than 1sec.
Figure 4.19: Minimum capture set slices at $x_3 = 0$ of transition starts from State 3 for three 3D car models. The minimum reach times for both cars are 1sec, we can infer that the actual minimum reach times for this transition are less than 1sec.

scheduling feature, when multi-core resources are available, the generated code will also adapt to the computation environment base on the user specification. In this example, we have four state transitions, both capture set and collision set need to be computed efficiently. Suppose we have 4-node computation capability, the computation will be scheduled as in table 3.2.7. The first computation will compute the minimum capture set with sparse grid setting, then the second computation will take the minimum capture time as the initial setting for the minimum capture set computation under dense grid setting, finally the collision set will be computed up to minimum capture time under dense grid setting.

The differences between sparse and dense computation grids are shown in Figure 4.21, Figure 4.22 and Figure 4.23. The solid line shapes in these figures gives a rough estimation for the minimum capture set which may not be accurate enough, and their transition time can be used to speed up the capture set computation with better accuracy. The dash dot line shapes are more accurate about the actual shapes
Figure 4.20: GME model for 3D UAV refueling example, this model is also built through model reusing.

Figure 4.21: Minimum capture set slices at $x_3 = 0$ of transition starts from State 1 with sparse and dense grids for UAV 1. Comparing with Step 2 capture sets, we can infer that the Step 1 capture set slice underestimates the actual capture set in all directions.
To observe different behaviors of different UAV models, three UAV models are verified. The specifications of three UAV models are as following:

<table>
<thead>
<tr>
<th>UAV Model Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Speed(m/s)</td>
<td>100</td>
<td>90</td>
<td>120</td>
</tr>
<tr>
<td>Minimum Speed(m/s)</td>
<td>60</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Maximum Turning Rate(deg/s)</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Turning Rate(deg/s)</td>
<td></td>
<td>-60</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: The specifications of three example UAV models.
Figure 4.22: Minimum capture set slices at $x_3 = 0$ of transition starts from State 2 with sparse and dense grids for UAV 1. The Target 1 Step 1 capture set slice has some overestimation in certain directions, and the Target 2 Step 1 capture set slice have overestimation in most directions.
Figure 4.23: Minimum capture set slices at $x_3 = 0$ of transition starts from State 3 with sparse and dense grids for UAV 1. The Step 1 capture set slice here has overestimation towards the bottom right direction.
Figure 4.24: Minimum capture set slices of transition starts from State 1 at $x_3 = 0$ for three 3D UAV models. The minimum reach times for these three UAVs are 2.86sec, 3.93sec and 2.29sec.
4.2.5 Improve Model Design Base on Reachable Sets

The safety check tool not only checks whether the system satisfy certain specifications, but also has the potential to improve the system structure design base on the previous reachable set results. Suppose we need to verify the system with a car that transition between different states, the protocol requires that the minimum reach time for each transition must be less than 10sec. The initial location of car is [5,5] and the target location is [20,45]. So the system can be modeled as in Figure 4.32, where there are only two States representing the initial state and target state and one TransitionMode connecting two states.

After verification, the results are as in Figure 4.33. As this design violates the protocol, base on the capture set result, we can insert more states between the initial state and target state to decrease the minimum reach time for each transition. The system can be modified as in Figure 4.34, two States([10, 25] and [15, 35]) are inserted between the initial state and final state, and there are three TransitionModes between the initial state and final state. After interpretation and verification, the verification result for TransitionMode1, TransitionMode2 and TransitionMode3 are as in Figure 4.35, Figure 4.36 and Figure 4.37. The minimum reach time of all three transitions are bellow 10sec, thus the new design is validated to be safe.

4.2.6 Computation Time Quantization

With low cost initial prediction, the total computation time can be dramatically reduced in most cases. For a 3D dynamical system, the total computation time will have \(O(N^3)\) complexity with respect to the grid number increase in Fig. 4.38. In the case of initial underestimation, the total computation time for parallel computation algorithm will have a slight increase as in Fig. 4.39. However, in the case of initial overestimation in Fig. 4.40, the parallel computation algorithm could dramatically decrease the computation time comparing to direct computation.
Figure 4.25: Minimum capture set slices of transition starts from State 2 at $x_3 = 0$ for three 3D UAV models. The minimum reach times for these three UAVs in top figure are 1.53sec, 2.93sec and 1.07sec. The minimum reach times for these three UAVs are 1.64sec, 3.59sec and 1.07sec.
Figure 4.26: Minimum capture set slices of transition starts from State 3 at $x_3 = 0$ for three 3D UAV models. The minimum reach times for these three UAVs are 5.43sec, 5.64sec and 6.93sec.
Figure 4.27: The capture set and collision set slices of three different UAVs in TransitionMode1 at $x_3 = 0$.

Figure 4.28: The capture set and collision set slices of three different UAVs in TransitionMode2 at $x_3 = 0$. 
Figure 4.29: The capture set and collision set slices of three different UAVs in TransitionMode4 at $x_3 = 0$.

Figure 4.30: The capture set and collision set slices of three different UAVs in TransitionMode3 at $x_3 = 0$. 
Figure 4.31: The unsafe set slices for different escape maneuvers at $x_3 = 0$.

Figure 4.32: The GME model for 3D car system with one transition.
Transition from State 1 [5,5] to State 2 [20,45] takes 10.43s

Figure 4.33: The capture set slices of transition1 at $x_3 = 0$. The minimum reach time is 10.43sec, which means this design violates the protocol.

Figure 4.34: The GME model for 3D car system with three transitions.
Transition from State 1 [5,5] to State 3 [10,25] takes 6.07s

Figure 4.35: The capture set slices of transition1(from [5, 5] to [10, 25]) at $x_3 = 0$ for updated model. The minimum reach time for this transition is 6.07sec.

Transition from State 3 [10,25] to State 4 [15,35] takes 5.88s

Figure 4.36: The capture set slices of transition2(from [10, 25] to [15, 35]) at $x_3 = 0$ for updated model. The minimum reach time for this transition is 5.88sec.
Transition from State 4 [15,35] to State 2 [20,45] takes 5.88s.

Figure 4.37: The capture set slices of transition3(from [15, 35] to [20, 45]) at $x_3 = 0$ for updated model. The minimum reach time for this transition is 5.88sec.

Figure 4.38: The total computation time for the same reachable set computation vs different grid number setting.
Comparison between Different Grid Number and Time Estimation

grid number 31 prediction
grid number 51 calculation
additional checks

Figure 4.39: In the case of initial underestimation, there will be a slight punishment in total computation time for parallel scheduling algorithm.

Comparison between Different Grid Number and Time Estimation

grid number 31 prediction
grid number 51 calculation
additional checks

Figure 4.40: In the case of overestimation, the parallel scheduling algorithm could save huge amount of computation time.
CHAPTER 5

Conclusion

5.1 Contribution

This dissertation proposes a tool for modeling, analysis and verification of multi-mode dynamical systems whose behavior can be described with Hamilton-Jacobi Equations. The embedded human abstraction is proposed to simplify the system modeling, and it also enables the computational aspects of system verification to be carried out in parallel, and for portions of the system model to be centrally abstracted, and easily replaced or reformulated as new models are introduced.

In the embedded human abstraction, there are only two types of system state: stationary state and transition state, a system in transition state will always end in a stationary state, and the transition from stationary state to transition state should be initiated by a human operator or designed as autonomous. DSM is used to constrain the modeling environment to satisfy constraints of embedded human abstraction. Further, this abstraction permits a fundamental decomposition that allows parallelization of individual transition motions.

By using a DSM approach, the tool is proved to be reusable and reconfigurable for a family of dynamical systems, and several different examples are discussed in the dissertation to demonstrate this desired software behavior. The software is shown to be effective in reusing a working model for other platforms (a working autonomous flight model can easily be reused for autonomous car model or a walking human model) and observing shared behaviors for multiple platforms. For example, a working 1D example can be easily reused and reconfigured for 2D or 3D examples by changing a small amount of system descriptions in the modeling environment, which not only ease the development difficulty but also reduces design error.

The approach also works for platforms with alternative specs if these platforms
have the same dynamics. In this case, the user can easily make modifications in the parameter aspect of the modeling environment. By observing shared behaviors for multiple platforms, a protocol across multiple platforms can be demonstrated, which shows how the modeling tool permits the approach to scale with the number of platforms in a nice way.

Besides that, the software also contains parallel scheduling algorithm to speed up the LSM Tool computation. The parallel scheduling is based on embedded human abstraction, where the system is decomposed into several transition maneuvers. The user can specify the hardware configuration to optimize the verification computation.

The computation code generated by the interpreter is able to check the system safety behaviors by examine the verification results. If the system property is not satisfied, the computation code will easily permit the “growing” of reachable set. For example, in the case of initial underestimation the transition time will be increased to make the capture set “grow”.

In order to speed up parallel computation times, each reachable set computation starts with a sparse grid computation to estimate the growth time of the set. While this may cause a small performance hit in the worst case, the benefit in most use cases is to dramatically reduce the total time to verify a protocol, and enable the automatic discovery of the reach set growth time—which was previously required to be known \textit{a priori} by the designer. Now that this time can be automatically discovered, it permits a new class of users to take advantage of these kinds of system verification tools.
5.2 Future Work

This research demonstrated a modeling and verification tool for a family of Embedded Human Systems (EHS) whose system dynamic behavior can be represented by Hamilton-Jacobi equations. The EHS are modeled as hybrid systems and then abstracted into several steady states and transitions among these states. In order to make the tool easily reconfigurable and reusable, Domain Specific Modeling (DSM) is implemented to develop the meta-model of the modeling environment, thus once the model is built under this environment it can be interpreted by a computer software to generate verification code. The interpreter takes advantage of parallel computation scheduling algorithm to speed up the system verification, users can set the number of parallel processes based on the capabilities of their computer hardware.

For future work, more desirable features can be incorporated into the current tool with careful modification of current metamodel and interpreter. Thus new functions can be brought into the current modeling environment without making the old model unusable. This capability will make the software evolution more efficient.

For example, if we want to add a new feature that limits the total number of transitions for each system, we can insert a boolean value to the specific model without causing any conflict. Analytical analysis can be implemented to the parallel scheduling algorithm, thus the strength and weakness of the algorithm can be precisely evaluated. Users will benefit from additional analysis of system properties when they are trying to set the initial computation parameters.

Additional future work includes automatically generating or modifying protocols based on their current satisfaction (or failure) of design parameters. Such an approach would permit a model transformation to take as input a new system class, and automatically modify an existing embedded human interaction protocol in order that the new system class (with its own dynamics) until it meets the original design goals.
APPENDIX A

Safety Check Model Interpreter source codes

Listing A.1: ReachSet.h

```cpp
1 #ifndef REACHSET_H
2 #define REACHSET_H
3
4 #include <string>
5 #include <vector>
6 #include <fstream>
7 #include <map>
8
9 class Constant
10 {
11 public:
12     Constant(std::string name, std::string value)
13         : name(name), value(value) {}
14
15     friend std::ostream& operator<<(std::ostream& out, const Constant& rhs)
16     {
17         out << rhs.name << " = " << rhs.value << ";";
18         return out;
19     }
20 protected:
21     std::string name;
22     std::string value;
23 };
24
25 class ReachSet
26 {
27 public:
28     ReachSet(std::string filename,
29                 std::string functionname,
30                 std::string t,
31                 std::string accuracy="medium",
32                 bool isSafe = true);
33     "ReachSet();
34
35     void write();
```
void setIsSafeSet (bool safe) { isSafe = safe; }
void setPartialFunc(std::string pf) { partialFunc = pf; }
void setHamFunc(std::string hf) { hamFunc = hf; }
void setController(std::string ctrl) { controller = ctrl; }
void setGridSize(int Nx) { gridSize = Nx; }
void setTarget(std::vector<std::string> values)
    { targetValues = values; }
void setPreviousTarget(std::vector<std::string> values)
    { previousTargetValues = values; }
void setTolerance(std::vector<std::string> values)
    { targetTolerance = values; }
void setTargetShapeText(std::string targetShape)
    { targetShapeText = targetShape; }
void addConstant(std::string variable, std::string constant)
    { constants[variable] = constant; }
void setDynamics(std::string d)
    { dynamics = d; }
void addStateVar(std::string variable, std::string descr = "(no descr provided)"
    { stateVariables.push_back(std::pair<std::string, std::string>(variable, descr)); }

std::string getFilename() const { return filename; }
std::string getFunctionname() const { return functionname; }
std::vector<std::string> getTolerance() const { return targetTolerance; }
std::string getTitle() const { return title; }
std::string getConstants() const { return setConstants(); }
std::string getController() const { return controller; }
std::string getDynamics() const { return dynamics; }
bool getIsSafe() const { return isSafe; }

protected:
    std::fstream file;
    std::string filename;
    std::string functionname;
    std::string title;
    bool isSafe;
    int gridSize;
    std::string accuracy;
    std::string partialFunc;
    std::string hamFunc;
    std::string controller;
    std::string dynamics;

    std::string getTargetString();
Listing A.2: SafetyCheck.h

```cpp
#include <fstream>

#define DEFAULT_ACCURACY "medium"

class ReachSet {
public:
    ReachSet(std::string n, std::string s, std::string d) : name(n), src(s), dst(d) {};
    std::string getName() { return name; };
    std::string getSrc() { return src; };
    std::string getDst() { return dst; };

private:
    std::string name;
    std::string src;
    std::string dst;

};
```

```cpp
#include "././SafetyCheck/SafetyCheck/SafetyCheckBonX.h"
#include <fstream>

#define DEFAULT_ACCURACY "medium"

class RouteNode {
public:
    RouteNode(std::string n, std::string s, std::string d) : name(n), src(s), dst(d) {};
    std::string getName() { return name; };
    std::string getSrc() { return src; };
    std::string getDst() { return dst; };
private:
    std::string name;
    std::string src;
    std::string dst;

};
```
```cpp
20  std::string name;
21  std::string src;
22  std::string dst;
23 };
24
25 class SafetyCheck
26 {
27  public:
28    SafetyCheck(BON::Project& project);
29    virtual ~SafetyCheck();
30
31  protected:
32    void generateCodes(BON::StateTransitionDiagram d,
33                        int count);
34    /* call methods to generate computation codes */
35    void visitTransitionMode(BON::TransitionMode tm,
36                               BON::StateTransitionDiagram d,
37                        int count);
38    /* compute the backward reachable set for transition mode */
39    void visitState(BON::State s, BON::StateTransitionDiagram d,
40                        int count);
41    /* compute the invariant set for steady states */
42    void visitController(BON::Controller c, BON::State s,
43                            BON::StateTransitionDiagram d, int count);
44    /* set the controller parameters and logic */
45    void getVariables(ReachSet* rs);
46    /* set state variable name and description
data for stateVariables vector */
47    void addConstants(ReachSet* rs,
48                        std::set<BON::Constant> constants);
49    /* set constants(location) information for ReachSet object */
50    void addPrevConstants(ReachSet* rs,
51                        std::set<BON::Constant> constants);
52    /* set constants(location) of previous state for ReachSet object */
53    void addGains(ReachSet* rs, std::set<BON::Gain> gains);
54    /* set controller gains for the ReachSet object */
55    void addTolerance(ReachSet* rs,
56                        std::set<BON::Tolerance> tolerance);
57    /* set controller tolerance for the ReachSet object */
58    void generateAll(BON::StateTransitionDiagram d, int count);
59    /* generate driver code for the system verification
60     and arrange the order of computations */
61    void generateParameterInit(BON::StateTransitionDiagram d);
```
void generateRouteTableInit(SCBon::StateTransitionDiagram d);
void generateManeuver(SCBon::StateTransitionDiagram d);
void addRouteTable(SCBon::StateTransitionDiagram d);
void addStates(SCBon::StateTransitionDiagram d);

std::vector<std::string> getTargetSets(ReachSet *rs, SCBon::TargetSet target);

std::vector<std::string> getPreviousTargetSets(ReachSet *rs, SCBon::TargetSet target);

std::vector<std::string> getTargetTolerance();

std::vector<std::string> getInvSets(ReachSet *rs, SCBon::InvSet inv);

std::string getTargetShapeText(SCBon::TargetSet target);

std::string getInvShapeText(SCBon::InvSet inv);

std::map<SCBon::StateTransitionDiagram, std::string> filenames;

std::map<SCBon::StateTransitionDiagram, std::vector<ReachSet>> reachsets;

std::map<SCBon::StateTransitionDiagram, std::vector<ReachSet>> escapesets;

std::vector<std::pair<std::string, std::string>> axes;

std::vector<SCBon::TargetSet> targetsets;

std::vector<SCBon::InvSet> invsets;

std::vector<RouteNode> routeTable;
std::vector<std::string> stateConstants;

/* store state constant names */
std::map<std::string, std::string> parameterConstants;

/* store parameter constant name and description */
};
APPENDIX B

Example visualization MATLAB scripts

By running script `multiModeGUIMain` as in listing B.1, the result of system verification can be visualized. The buttons of User Interface are transition initialization button, escape initialization button and pause button as in Figure B.1. There is also a toggle button at the upper left corner for user to grey unsafe escape buttons.

When the verification computation is finished, capture set data (collision set data if they are specified) of each time step for all transition states are stored into .mat files for the later visualization. If escape modes are added to the system model, the visualization will have three types of states: steady state, transition state and escape state. The visualization state diagram is as in Figure B.2. When the visualization is started, the initial steady state will be displayed to the user, as in Figure B.1 all transitions that are validated by the capture sets (and collision sets if they are specified) are displayed on the interface. Figure B.1 is the visualization for the 3D example 2(UAV refueling example) as in Section 4.2.4, the displayed state is the initial steady state $s_1$, and only one target state $s_2$ for $s_1$ is shown with symbol $o$, and the name and symbol for transition between state $s_1$ and $s_2$ are shown on the button to initialize the transition. Different escape maneuver initialization buttons are placed at the right side of the interface. As in Figure B.2, we can leave Steady State and go to Transition State or Escape State, as only safety guaranteed transition buttons are displayed on the interface, no unsafe decision will be made by the user. User can also enter the Escape State from the Transition State, however once the user enters the Escape State, the system will always stay in Escape State.

The MATLAB functions behind the visualization are organized as Figure B.3, all model-dependent scripts that are inside the dashed line rectangle are generated by the interpreter.
Figure B.1: The User Interface for verification results. All safety guaranteed transition states are displayed on the interface, there are one transition mode(TransitionMode1,o) and five escape modes(Turn right, Turn left, Speed up, Slow down and Quick escape). The Pause button is at the lower right corner, by pressing the Pause button all state will be paused and the state will continue once this button is pressed again. The collision sets for transition states and escape states and capture sets for transition states are shown by different color and line types. Solid red line and solid green line are moving trajectories of UAV and tanker. The blue shape is the capture set slices for TransitionMode1 and the its related collision set slice is shown by the magenta shape, the target state is shown with symbol o. Different types of red shapes are collision sets for five escape modes. The black shape shows the minimum separation set of the tanker.
State Diagrams

Figure B.2: The state transition diagram for result visualization program. The system will start from the initial steady state and end when there is no transition state available for the steady state the system is targeting at.

Function Calling Graph

Figure B.3: The function calling graph for result visualization program. All model-dependent scripts that are inside the dashed line rectangle are generated by the interpreter. The data from reachable set computation are represented with an elliptical shape. All model-independent functions are placed outside the dashed line rectangle.
Listing B.1: multiModeGUIMain.m

```matlab
function varargout = multiModeGUIMain(varargin)

% MULTIODEGUIMAIN MATLAB code for multiModeGUIMain.fig
% MULTIODEGUIMAIN, by it self, creates a new MULTIODEGUIMAIN or raises the existing
% singleton.
%
% H = MULTIODEGUIMAIN returns the handle to a new MULTIODEGUIMAIN or the handle to
% the existing singleton.
%
% MULTIODEGUIMAIN('CALLBACK', hObject, eventData, handles, ...) calls the local
% function named CALLBACK in MULTIODEGUIMAIN.M with the given input arguments.
%
% MULTIODEGUIMAIN('Property', 'Value', ...) creates a new MULTIODEGUIMAIN or raises the
% existing singleton*. Starting from the left, property value pairs are
% applied to the GUI before multiModeGUIMain_OpeningFcn gets called. An
% unrecognized property name or invalid value makes property application
% stop. All inputs are passed to multiModeGUIMain_OpeningFcn via varargin.
%
% *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only one
% instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help multiModeGUIMain

% Last Modified by GUIDE v2.5 23–Apr–2012 17:06:55

% Begin initialization code – DO NOT EDIT

gui_Singleton = 1;

% gui_State = struct ('gui_Name', mfilename, ...
% 'gui_Singleton', gui_Singleton, ...
% 'gui_OpeningFcn', @multiModeGUIMain_OpeningFcn, ...
% 'gui_OutputFcn', @multiModeGUIMain_OutputFcn, ...
% 'gui_LayoutFcn', [], ...
% 'gui_Callback', []);

if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end

% End initialization code – DO NOT EDIT
```
% -- Executes just before multiModeGUIMain is made visible.

function multiModeGUIMain_OpeningFcn(hObject, eventdata, handles, varargin)

% This function has no output args, see OutputFcn.

% hObject handle to figure
% eventdata reserved -- to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
% varargin command line arguments to multiModeGUIMain (see VARARGIN)

% Choose default command line output for multiModeGUIMain
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes multiModeGUIMain wait for user response (see Uiresume)
% uiwait(hObject.

% Outputs from this function are returned to the command line.

function varargout = multiModeGUIMain_OutputFcn(hObject, eventdata, handles)

% varargout cell array for returning output args (see varargout)
% hObject handle to figure
% eventdata reserved -- to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

global s_flag t_flag e_flag;

global tankPos;

global routeTable;


global currentRoute;


global waypoints;


global gList esg;


global rsData rsTime usData usTime esData esTime;


global specific_theta;


global waitTime;


global hObjectCopy;


global handlesCopy;

addpath data\

set(handles.pushbutton1,'Visible','OFF');
set(handles.pushbutton2,'Visible','OFF');
set(handles.pushbutton3,'Visible','OFF');
set(handles.pushbutton4,'Enable','OFF','String','Turn right');
set(handles.pushbutton5,'Enable','OFF','String','Turn left');
set(handles.pushbutton6,'Enable','OFF','String','Speed up');
set(handles.pushbutton7,'Enable','OFF','String','Slow down');
set(handles.pushbutton8,'Enable','OFF','String','Quick escape');
set(handles.checkbox1,'String','Grey unsafe escape');
set(handles.togglebutton1,'String','Pause');
hObjectCopy = hObject;
handlesCopy = handles;
setappdata(hObjectCopy,'cbr0',1);
s_flag = 1;
t_flag = 1;
e_flag = 0;
tankPos = [0 0];
specific_theta = 0;
waitTime = 10;
routeTableInit
currentRoute = find(routeTable(:,1)==1);
	[tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ... 
	loadReachsets(currentRoute(1));
tCount = length(tname);
switch tCount
    case 1
        set(handles.pushbutton1,'Visible','ON','String',tname{1});
send
    case 2
        set(handles.pushbutton1,'Visible','ON','String',tname{1});
        set(handles.pushbutton2,'Visible','ON','String',tname{2});
send
    case 3
        set(handles.pushbutton1,'Visible','ON','String',tname{1});
        set(handles.pushbutton2,'Visible','ON','String',tname{2});
        set(handles.pushbutton3,'Visible','ON','String',tname{3});
    otherwise
        sprintf('There is something wrong with the transition name.');
end
if ~isempty(esData)
    set(handles.pushbutton4,'Enable','ON');
    set(handles.pushbutton5,'Enable','ON');
    set(handles.pushbutton6,'Enable','ON');
    set(handles.pushbutton7,'Enable','ON');
    set(handles.pushbutton8,'Enable','ON');
while ((inTime < waitTime) && (getappdata(hObjectCopy, 'cbr0') == 1))
    [tankPos inTime] = displaySteady(inTime, tankPos, waypoints, ... 
        gList, rsData, rsTime, ... 
        usData, usTime, esg, esData, esTime, specific_theta);
end

% -- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
    global s_flag t_flag e_flag;
    global tankPos;
    global routeTable;
    global currentRoute;
    global waypoints;
    global gList esg;
    global rsData rsTime usData usTime esData esTime;
    global specific_theta;
    global waitTime;
    global hObjectCopy;

    setappdata(hObjectCopy, 'cbr1', 1);
    [tankPos] = displayTransition(tankPos, ... 
        routeTable(previousRoute, 2), gList, rsData, rsTime, ... 
        currentRoute = find(routeTable(:,1) == routeTable(currentRoute(1), 2));
    t_flag = length(currentRoute);

    setappdata(hObjectCopy, 'cbr1', 1);
usData, usTime, specific_theta);

if (t_flag == 3)
    set (handles.pushbutton1, 'Enable', 'ON');
    set (handles.pushbutton2, 'Enable', 'ON');
    set (handles.pushbutton3, 'Enable', 'ON');
else if (t_flag == 2)
    set (handles.pushbutton1, 'Enable', 'ON');
    set (handles.pushbutton2, 'Enable', 'ON');
    set (handles.pushbutton3, 'Enable', 'OFF');
else if (t_flag == 1)
    set (handles.pushbutton1, 'Enable', 'ON');
    set (handles.pushbutton2, 'Enable', 'OFF');
    set (handles.pushbutton3, 'Enable', 'OFF');
else
    set (handles.pushbutton1, 'Enable', 'OFF');
    set (handles.pushbutton2, 'Enable', 'OFF');
    set (handles.pushbutton3, 'Enable', 'OFF');
end
end
end
inTime = 0;

if (t_flag ~= 0)
    clear rsData usData esData
    [tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
    loadReachsets(currentRoute(1));
    tCount = length(tname);
    switch tCount
        case 1
            set (handles.pushbutton1, 'Visible', 'ON', 'String', tname{1});
        case 2
            set (handles.pushbutton1, 'Visible', 'ON', 'String', tname{1});
            set (handles.pushbutton2, 'Visible', 'ON', 'String', tname{2});
        case 3
            set (handles.pushbutton1, 'Visible', 'ON', 'String', tname{1});
            set (handles.pushbutton2, 'Visible', 'ON', 'String', tname{2});
            set (handles.pushbutton3, 'Visible', 'ON', 'String', tname{3});
        otherwise
            sprintf('There is something wrong with the transition name.');
        end
    set (handles.pushbutton4, 'Enable', 'ON');
    set (handles.pushbutton5, 'Enable', 'ON');
    set (handles.pushbutton6, 'Enable', 'ON');
    set (handles.pushbutton7, 'Enable', 'ON');
    while ((inTime < waitTime) && (getappdata(hObjectCopy, 'cbr1') == 1))
[tankPos inTime ] = displaySteady(inTime, tankPos, waypoints, ...
  gList, rsData, rsTime, ...
  usData, usTime, esg, esData, esTime, specific_theta);
end
end

setappdata(hObjectCopy, 'cbr0', 0);
setappdata(hObjectCopy, 'cbr1', 0);
setappdata(hObjectCopy, 'cbr2', 0);
setappdata(hObjectCopy, 'cbr3', 0);
s_flag = s_flag+1;

% ---- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
  % hObject    handle to pushbutton2 (see GCBO)
  % eventdata  reserved - to be defined in a future version of MATLAB
  % handles    structure with handles and user data (see GUIDATA)
  global s_flag t_flag e_flag;
  global tankPos;
  global routeTable;
  global currentRoute;
  global waypoints;
  global gList esg;
  global rsData rsTime usData usTime esData esTime;
  global specific_theta;
  global waitTime;
  global hObjectCopy;

  set(handles.pushbutton1, 'Visible', 'OFF');
  set(handles.pushbutton2, 'Visible', 'OFF');
  set(handles.pushbutton3, 'Visible', 'OFF');
  set(handles.pushbutton4, 'Enable', 'OFF');
  set(handles.pushbutton5, 'Enable', 'OFF');
  set(handles.pushbutton6, 'Enable', 'OFF');
  set(handles.pushbutton7, 'Enable', 'OFF');

  previousRoute = currentRoute(2);
  currentRoute = find(routeTable(:,1)==routeTable(currentRoute(2), 2));
  t_flag = length(currentRoute);
[tankPos] = displayTransition(tankPos,...
routeTable(previousRoute,2),gList,rsData,rsTime,...
usData,usTime,specific_theta);
if(t_flag==3)
    set(handles.pushbutton1,'Enable','ON');
    set(handles.pushbutton2,'Enable','ON');
    set(handles.pushbutton3,'Enable','ON');
else if(t_flag==2)
    set(handles.pushbutton1,'Enable','ON');
    set(handles.pushbutton2,'Enable','ON');
    set(handles.pushbutton3,'Enable','OFF');
else if(t_flag==1)
    set(handles.pushbutton1,'Enable','OFF');
    set(handles.pushbutton2,'Enable','OFF');
    set(handles.pushbutton3,'Enable','OFF');
else
    set(handles.pushbutton1,'Enable','OFF');
    set(handles.pushbutton2,'Enable','OFF');
    set(handles.pushbutton3,'Enable','OFF');
end
end
end
inTime = 0;
if(t_flag ~= 0)
    clear rsData usData esData
    [tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
    loadReachsets(currentRoute(1));
    tCount = length(tname);
    switch tCount
    case 1
        set(handles.pushbutton1,'Visible','ON','String',tname{1});
    case 2
        set(handles.pushbutton1,'Visible','ON','String',tname{1});
        set(handles.pushbutton2,'Visible','ON','String',tname{2});
    case 3
        set(handles.pushbutton1,'Visible','ON','String',tname{1});
        set(handles.pushbutton2,'Visible','ON','String',tname{2});
        set(handles.pushbutton3,'Visible','ON','String',tname{3});
    otherwise
        sprintf('There is something wrong with the transition name.');
    end
    set(handles.pushbutton4,'Enable','ON');
    set(handles.pushbutton5,'Enable','ON');
    set(handles.pushbutton6,'Enable','ON');
    set(handles.pushbutton7,'Enable','ON');
while ((inTime < waitTime) && (getappdata(hObjectCopy, 'cbr2') == 1))
        [tankPos inTime] = displaySteady(inTime, tankPos, waypoints, ...
        gList, rsData, rsTime, ...
        usData, usTime, esg, esData, esTime, specific_theta);
    end
end

setappdata(hObjectCopy, 'cbr0', 0);
setappdata(hObjectCopy, 'cbr1', 0);
setappdata(hObjectCopy, 'cbr2', 0);
setappdata(hObjectCopy, 'cbr3', 0);
s_flag = s_flag + 1;

% −−− Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
    % hObject handle to pushbutton3 (see GCBO)
    % eventdata reserved – to be defined in a future version of MATLAB
    % handles structure with handles and user data (see GUIDATA)
    global s_flag t_flag e_flag;
    global tankPos;
    global routeTable;
    global waypoints;
    global gList esg;
    global rsData rsTime usData usTime esData esTime;
    global specific_theta;
    global waitTime;
    global hObjectCopy;

    set(handles.pushbutton1, 'Visible', 'OFF');
    set(handles.pushbutton2, 'Visible', 'OFF');
    set(handles.pushbutton3, 'Visible', 'OFF');
    set(handles.pushbutton4, 'Enable', 'OFF');
    set(handles.pushbutton5, 'Enable', 'OFF');
    set(handles.pushbutton6, 'Enable', 'OFF');
    set(handles.pushbutton7, 'Enable', 'OFF');
    previousRoute = currentRoute(3);

    currentRoute = find(routeTable(:,1)==routeTable(currentRoute(3),2));
    t_flag = length(currentRoute);
    setappdata(hObjectCopy, 'cbr3', 1);
    [tankPos] = displayTransition(tankPos, ...
routeTable (previousRoute, 2), gList, rsData, rsTime, ...
usData, usTime, specific_theta);

if (t_flag == 3)  
    set (handles.pushbutton1, 'Enable', 'ON');
    set (handles.pushbutton2, 'Enable', 'ON');
    set (handles.pushbutton3, 'Enable', 'ON');
else if (t_flag == 2)  
    set (handles.pushbutton1, 'Enable', 'ON');
    set (handles.pushbutton2, 'Enable', 'ON');
    set (handles.pushbutton3, 'Enable', 'OFF');
else if (t_flag == 1)  
    set (handles.pushbutton1, 'Enable', 'OFF');
    set (handles.pushbutton2, 'Enable', 'OFF');
    set (handles.pushbutton3, 'Enable', 'OFF');
else
    set (handles.pushbutton1, 'Enable', 'OFF');
    set (handles.pushbutton2, 'Enable', 'OFF');
    set (handles.pushbutton3, 'Enable', 'OFF');
end
end
end
inTime = 0;
if (t_flag ~ 0)
    clear rsData usData esData
    [tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
    loadReachsets (currentRoute (1));
    tCount = length (tname);
    switch tCount
        case 1
    set (handles.pushbutton1, 'Visible', 'ON', 'String', tname{1});
        case 2
    set (handles.pushbutton1, 'Visible', 'ON', 'String', tname{1});
    set (handles.pushbutton2, 'Visible', 'ON', 'String', tname{2});
        case 3
    set (handles.pushbutton1, 'Visible', 'ON', 'String', tname{1});
    set (handles.pushbutton2, 'Visible', 'ON', 'String', tname{2});
    set (handles.pushbutton3, 'Visible', 'ON', 'String', tname{3});
        otherwise
    sprintf ('There is something wrong with the transition name.');
    end
    set (handles.pushbutton4, 'Enable', 'ON');
    set (handles.pushbutton5, 'Enable', 'ON');
    set (handles.pushbutton6, 'Enable', 'ON');
    set (handles.pushbutton7, 'Enable', 'ON');
while ((inTime < waitTime) && (getappdata(hObjectCopy, 'cbr3') == 1))
[tankPos inTime] = displaySteady(inTime, tankPos, waypoints, ...
gList, rsData, rsTime, ...
usData, usTime, esg, esData, esTime, specific_theta);
end

setappdata(hObjectCopy, 'cbr0', 0);
setappdata(hObjectCopy, 'cbr1', 0);
setappdata(hObjectCopy, 'cbr2', 0);
setappdata(hObjectCopy, 'cbr3', 0);
s_flag = s_flag+1;

function pushbutton4_Callback(hObject, eventdata, handles)
  global s_flag t_flag e_flag;
global tankPos;
global routeTable;
global currentRoute;
global waypoints;
global gList esg;
global rsData rsTime usData usTime esData esTime;
global specific_theta;
global waitTime;
global hObjectCopy;
clear rsData usData esData
[tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
  loadReachsets(currentRoute(1));

set(handles.pushbutton1, 'Visible', 'OFF');
set(handles.pushbutton2, 'Visible', 'OFF');
set(handles.pushbutton3, 'Visible', 'OFF');
set(handles.pushbutton4, 'Enable', 'OFF');
set(handles.pushbutton5, 'Enable', 'OFF');
set(handles.pushbutton6, 'Enable', 'OFF');
set(handles.pushbutton7, 'Enable', 'OFF');
[tankPos] = displayEscape(tankPos ,1 ,esg ,esData ,esTime ,specific_theta)

setappdata(hObjectCopy, 'cbr0', 0);
setappdata(hObjectCopy, 'cbr1', 0);
setappdata(hObjectCopy, 'cbr2', 0);
setappdata(hObjectCopy, 'cbr3', 0);
setappdata(hObjectCopy, 'cbr4', 0);
450 setappdata(hObjectCopy,'cbr5',0);
451 setappdata(hObjectCopy,'cbr6',0);
452 setappdata(hObjectCopy,'cbr7',0);
453 setappdata(hObjectCopy,'cbr8',0);
454 e_flag = 1;
455 set(handles.pushbutton4,'Enable','ON');
456 set(handles.pushbutton5,'Enable','ON');
457 set(handles.pushbutton6,'Enable','ON');
458 set(handles.pushbutton7,'Enable','ON');
459 setappdata(hObjectCopy,'cbr4',1);
460 inTime = 0;
461 while ((inTime < waitTime) && (getappdata(hObjectCopy,'cbr4') == 1))
462    [tankPos inTime] = displaySteady(inTime,tankPos,waypoints,...
463        gList, rsData, rsTime,...
464        usData, usTime, esg, esData, esTime, specific_theta);
465 end
466
467 % ---- Executes on button press in pushbutton5.
468 function pushbutton5_Callback(hObject, eventdata, handles)
469 % hObject handle to pushbutton5 (see GCBO)
470 % eventdata reserved - to be defined in a future version of MATLAB
471 % handles structure with handles and user data (see GUIDATA)
472 global s_flag t_flag e_flag;
473 global tankPos;
474 global routeTable;
475 global currentRoute;
476 global waypoints;
477 global gList esg;
478 global rsData rsTime usData usTime esData esTime;
479 global specific_theta;
480 global waitTime;
481 global hObjectCopy;
482 clear rsData usData esData
483 [tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
484    loadReachsets(currentRoute(1));
485
486 set(handles.pushbutton1,'Visible','OFF');
487 set(handles.pushbutton2,'Visible','OFF');
488 set(handles.pushbutton3,'Visible','OFF');
489 set(handles.pushbutton4,'Enable','OFF');
490 set(handles.pushbutton5,'Enable','OFF');
491 set(handles.pushbutton6,'Enable','OFF');
492 set(handles.pushbutton7,'Enable','OFF');
493 [tankPos] = displayEscape(tankPos,2, esg, esData, esTime, specific_theta);
setappdata(hObjectCopy,'cbr0',0);
setappdata(hObjectCopy,'cbr1',0);
setappdata(hObjectCopy,'cbr2',0);
setappdata(hObjectCopy,'cbr3',0);
setappdata(hObjectCopy,'cbr4',0);
setappdata(hObjectCopy,'cbr5',0);
setappdata(hObjectCopy,'cbr6',0);
setappdata(hObjectCopy,'cbr7',0);
setappdata(hObjectCopy,'cbr8',0);
e_flag = 1;
set(handles.pushbutton4,'Enable','ON');
set(handles.pushbutton5,'Enable','ON');
set(handles.pushbutton6,'Enable','ON');
set(handles.pushbutton7,'Enable','ON');
setappdata(hObjectCopy,'cbr5',1);
inTime = 0;
while ((inTime < waitTime) && (getappdata(hObjectCopy,'cbr5') == 1))
    [tankPos inTime] = displaySteady(inTime,tankPos,waypoints,...
gList,rsData,rsTime,...
    usData,usTime,esg,esData,esTime,specific_theta);
end

% -- Executes on button press in pushbutton6.
function pushbutton6_Callback(hObject, eventdata, handles)
% hObject handle to pushbutton6 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
global s_flag t_flag e_flag;
global tankPos;
global routeTable;
global currentRoute;
global waypoints;
global gList,esg;
global rsData rsTime usData usTime esData esTime;
global specific_theta;
global waitTime;
global hObjectCopy;
clear rsData usData esData
[tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
loadReachsets(currentRoute(1));
set(handles.pushbutton1,'Visible','OFF');
set(handles.pushbutton2,'Visible','OFF');
set(handles.pushbutton3,'Visible','OFF');
set(handles.pushbutton4,'Enable','OFF');
set(handles.pushbutton5,'Enable','OFF');
set(handles.pushbutton6,'Enable','OFF');
set(handles.pushbutton7,'Enable','OFF');
[tankPos] = displayEscape(tankPos,3,esg,esData,esTime,specific_theta);
setappdata(hObjectCopy,'cbr0',0);
setappdata(hObjectCopy,'cbr1',0);
setappdata(hObjectCopy,'cbr2',0);
setappdata(hObjectCopy,'cbr3',0);
setappdata(hObjectCopy,'cbr4',0);
setappdata(hObjectCopy,'cbr5',0);
setappdata(hObjectCopy,'cbr6',0);
setappdata(hObjectCopy,'cbr7',0);
setappdata(hObjectCopy,'cbr8',0);
e_flag = 1;
set(handles.pushbutton4,'Enable','ON');
set(handles.pushbutton5,'Enable','ON');
set(handles.pushbutton6,'Enable','ON');
set(handles.pushbutton7,'Enable','ON');
setappdata(hObjectCopy,'cbr6',1);
inTime = 0;
while((inTime < waitTime) && (getappdata(hObjectCopy,'cbr6')==1))
    [tankPos inTime] = displaySteady(inTime,tankPos,waypoints,...
    gList,rsData,rsTime,...
    usData,usTime,esg,esData,esTime,specific_theta);
end

% ---- Executes on button press in pushbutton7.
function pushbutton7_Callback(hObject, eventdata, handles)
    hObject handle to pushbutton7 (see GCBO)
    eventdata reserved -- to be defined in a future version of MATLAB
    handles structure with handles and user data (see GUIDATA)
    global s_flag t_flag e_flag;
    global tankPos;
    global routeTable;
    global currentRoute;
    global waypoints;
    global gList esg;
    global rsData rsTime usData usTime esData esTime;
    global specific_theta;
    global waitTime;
    global hObjectCopy;
    clear rsData usData esData
    [tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
set(handles.pushbutton1,'Visible','OFF');
set(handles.pushbutton2,'Visible','OFF');
set(handles.pushbutton3,'Visible','OFF');
set(handles.pushbutton4,'Enable','OFF');
set(handles.pushbutton5,'Enable','OFF');
set(handles.pushbutton6,'Enable','OFF');
set(handles.pushbutton7,'Enable','OFF');
[tankPos] = displayEscape(tankPos,4,esg,esData,esTime,specific_theta);
setappdata(hObjectCopy,'cbr0',0);
setappdata(hObjectCopy,'cbr1',0);
setappdata(hObjectCopy,'cbr2',0);
setappdata(hObjectCopy,'cbr3',0);
setappdata(hObjectCopy,'cbr4',0);
setappdata(hObjectCopy,'cbr5',0);
setappdata(hObjectCopy,'cbr6',0);
setappdata(hObjectCopy,'cbr7',0);
setappdata(hObjectCopy,'cbr8',0);
e_flag = 1;
set(handles.pushbutton4,'Enable','ON');
set(handles.pushbutton5,'Enable','ON');
set(handles.pushbutton6,'Enable','ON');
set(handles.pushbutton7,'Enable','ON');
setappdata(hObjectCopy,'cbr7',1);
inTime = 0;
while((inTime<waitTime)&(getappdata(hObjectCopy,'cbr7')==1))
    [tankPos inTime] = displaySteady(inTime,tankPos,waypoints,...
        gList,rsData,rsTime,...
        usData,usTime,esg,esData,esTime,specific_theta);
end

% ---- Executes on button press in pushbutton8.
function pushbutton8_Callback(hObject eventdata handles)
% hObject handle to pushbutton8 (see GCBO)
% eventdata reserved – to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
global s_flag t_flag e_flag;
global tankPos;
global routeTable;
global currentRoute;
global waypoints;
global gList esg;
global rsData rsTime usData usTime esData esTime;
global specific_theta;
global waitTime;
global hObjectCopy;
clear rsData usData esData
tname waypoints gList rsData rsTime usData usTime esg esData esTime] = ...
loadReachsets(currentRoute(1));

set(handles.pushbutton1,'Visible','OFF');
set(handles.pushbutton2,'Visible','OFF');
set(handles.pushbutton3,'Visible','OFF');
set(handles.pushbutton4,'Enable','OFF');
set(handles.pushbutton5,'Enable','OFF');
set(handles.pushbutton6,'Enable','OFF');
set(handles.pushbutton7,'Enable','OFF');
set(handles.pushbutton8,'Enable','OFF');
[tankPos] = displayEscape(tankPos,5,esg,esData,esTime,specific_theta);

setappdata(hObjectCopy,'cbr0',0);
setappdata(hObjectCopy,'cbr1',0);
setappdata(hObjectCopy,'cbr2',0);
setappdata(hObjectCopy,'cbr3',0);
setappdata(hObjectCopy,'cbr4',0);
setappdata(hObjectCopy,'cbr5',0);
setappdata(hObjectCopy,'cbr6',0);
setappdata(hObjectCopy,'cbr7',0);
setappdata(hObjectCopy,'cbr8',0);
e_flag = 1;
set(handles.pushbutton4,'Enable','ON');
set(handles.pushbutton5,'Enable','ON');
set(handles.pushbutton6,'Enable','ON');
set(handles.pushbutton7,'Enable','ON');
set(handles.pushbutton8,'Enable','ON');
setappdata(hObjectCopy,'cbr8',1);
inTime = 0;
while ((inTime < waitTime) && (getappdata(hObjectCopy,'cbr8') == 1))
[tankPos inTime ] = displaySteady(inTime,tankPos,waypoints,...
gList,rsData,rsTime,...
usData,usTime,esg,esData,esTime,specific_theta);
end

% ---- Executes on button press in checkbox1.
function checkbox1_Callback(hObject, eventdata, handles)
% hObject handle to checkbox1 (see GCBO)
% eventdata reserved – to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
Listing B.2: loadReachsets.m

```matlab
function [transitionName waypointsOut gList rsData rsTime ... usData usTime esg esData esTime] = loadReachsets(i)

if ( nargin<1)
i=1;
end

routeTableInit

[rowRT colRT] = size(routeTable);
if( exist(gInfo1))
iResult1 = load(gInfo1);
pType = 1;
end
if( exist(gInfo2))
iResult2 = load(gInfo2);
iResult1 = iResult2;
pType = 2;
end
if( exist(gInfo3))
iResult3 = load(gInfo3);
pType = 3;
esg = iResult3.reachsets.reachset_vector(1).g;
else
    esg = {};
esData = {};
esTime = {};
end
iFile1 = load(strcat(nameTable{i},'data.mat'));
iTime1 = load(strcat(nameTable{i},'time.mat'));
data1 = iFile1.dataList;
timel = iTime1.timeList;
```
gr = iResult1.reachsets.reachset_vector(i).g;
if exist(strcat(nameTable{i+rowRT},'data.mat'))
ifile2 = load(strcat(nameTable{i+rowRT},'data.mat'));
itime2 = load(strcat(nameTable{i+rowRT},'time.mat'));
data2 = ifile2.dataList;
time2 = itime2.timeList;
gu = iResult2.reachsets.reachset_vector(i+rowRT).g;
else
usData = {};
usTime = {};
end
transitionName{1} = ...
strcat(iResult1.reachsets.reachset_vector(i).title,'(o)');

locx1 = waypoints(routeTable(i,2),1);
locx1_prev = waypoints(routeTable(i,1),1);
locx2 = waypoints(routeTable(i,2),2);
locx2_prev = waypoints(routeTable(i,1),2);
locx3 = waypoints(routeTable(i,2),3);
locx3_prev = waypoints(routeTable(i,1),3);

waypointsOut = [locx1_prev locx2_prev locx3_prev;...
locx1 locx2 locx3];

gList{1} = gr;
if exist('gu')
gList{2} = gu;
usData{1} = data2;
usTime{1} = time2;
else
gList{2} = {};
end

rsData{1} = data1;
rsTime{1} = timel;

dotTable = {'+*'};
dotCount = 1;

if(length(find(routeTable(:,1) == routeTable(i,1))) > 1)
    for j = 1:rowRT
        if j==i & routeTable(j,1)==routeTable(i,1)
            ifile1 = load(strcat(nameTable{j},'data.mat'));
iTimel = load(strcat(nameTable{j},'time.mat'));
        end
    end
end

data1 = iFile1.dataList;
timel = iTimel.timeList;
gr = iResult1.reachsets.reachset_vector(j).g;

transitionName = [transitionName; ...
    strcat(iResult1.reachsets.reachset_vector(j).title,...
    dotTable{dotCount}]);
dotCount = dotCount + 1;
gList{end+1} = gr;
rsData{end+1} = data1;
rsTime{end+1} = timel;
    if exist(strcat(nameTable{j+rowRT},'data.mat'))
iFile2 = load(strcat(nameTable{j+rowRT},'data.mat'));
iTime2 = load(strcat(nameTable{j+rowRT},'time.mat'));
data2 = iFile2.dataList;
time2 = iTimel.timeList;
gu = iResult2.reachsets.reachset_vector(j+rowRT).g;
gList{end+1} = gu;
usData{end+1} = data2;
usTime{end+1} = time2;
else
    gList{end+1} = {};
end
locx1_new = waypoints(routeTable(j,2),1);
locx2_new = waypoints(routeTable(j,2),2);
locx3_new = 0;
waypointsOut = [waypointsOut; locx1_new locx2_new locx3_new];
end
end

if pType == 3
for k=1:5
    iFile3 = load(strcat('s',num2str(routeTable(i,1)),'...
    ','escape',num2str(k),'data.mat'));
iTime3 = load(strcat('s',num2str(routeTable(i,1)),'...
    ','escape',num2str(k),'time.mat'));
esData{k} = iFile3.dataList;
esTime{k} = iTimel.timeList;
end
end

Listing B.3: calcContours.m

function contours = calcContours( contourIndices )
Listing B.4: displaySteady.m

function [location outTime movieFrames ] = displaySteady(inTime, tankPosition, waypoints, ...
gList, rsData, rsTime, ...
usData, usTime, esg, esData, esTime, specific_theta) 

global handlesCopy;

plotCenter = [0 0];
plotSize = 100;
linetypeTable = {’-’ ’--’ ’-.’ ’;’};
dotTable = {’k.’ ’ko’ ’k+’ ’k*’};

loc1 = waypoints(2,1);
loc2 = waypoints(2,2);
loc3 = waypoints(2,3);

loc1_prev = waypoints(1,1);
loc2_prev = waypoints(1,2);
loc3_prev = 0;

parameterInit

[uavx,uavv,uavc] = uavpatch;
[tx,ty,tc] = tankerpatch;

scale=1;
uavx = scale*uavx;
uavv = scale*uavv;
tx = scale*tx;
ty = scale*ty;
tanker_x_offset = 15;

tx = tx + tanker_x_offset;
t_step = 0.05;
startLocation = tankPosition;
cIter = 1;
cMax = 3;

[rowLoc colLoc] = size(waypoints);

if nargin < 12
    specific_theta = 0;
end

ref = -1;
if ~exist('vmax_tanker')
    vmax_tanker = 0;
    ref = 1;
end

while (cIter < cMax)
    cLoc = 1;
    while (cLoc <= rowLoc)
        plot(ref*waypoints(cLoc,1)+startLocation(1)+vmax_tanker*t_step*cIter,...
             ref*waypoints(cLoc,2)+startLocation(2),dotTable{cLoc});
        hold on;
        cLoc = cLoc + 1;
    end
    if (nargin > 8 && size(esData,1)>0)
        for i=1:4
            ge = esg;
            indexe = ceil(ge.N(3)*(specific_theta+1)/2+1);
            x1_ge = linspace(ge.min(1),ge.max(1),ge.N(1));
            x2_ge = linspace(ge.min(2),ge.max(2),ge.N(2));
            esdata = esData{i}{end};
            escapeset = contourc(x1_ge,x2_ge,esdata(:,indexe),',0,0');
            contours=calcContours(escapeset);
            for j=1:length(contours)
                if ( j == length(contours) )
                    levelx=ref*escapeset(1,contours(j)+1:end);
                    leyley=ref*escapeset(2,contours(j)+1:end);
                else
                    levelx=ref*escapeset(1,contours(j)+1:contours(j+1)-1);
                    leyley=ref*escapeset(2,contours(j)+1:contours(j+1)-1);
                end
                levelx_abs = levelx + startLocation(1) + vmax_tanker*t_step*cIter;
                leyley_abs = leyley + startLocation(2);
                plot(levelx_abs,leyley_abs,linetypeTable{i},'color','r');
            end
        end
    end
156

80 end
81 ge = esg;
82 indexe = ceil((ge.N(3)*(specific_theta+1))/2+1);
83 x1_ge = linspace(ge.min(1),ge.max(1),ge.N(1));
84 x2_ge = linspace(ge.min(2),ge.max(2),ge.N(2));
85 esdata = esData{5}{end};
86 escapese = contourc(x1_ge,x2_ge,esdata(:,:,indexe)',[0,0]);
87 contours=calcContours(escapese);
88 for j=1:length(contours)
89 if ( j == length(contours) )
90 levelxe=ref*escapese(1,contours(j)+1:end);
91 levelye=ref*escapese(2,contours(j)+1:end);
92 else
93 levelxe=ref*escapese(1,contours(j)+1:contours(j+1)-1);
94 levelye=ref*escapese(2,contours(j)+1:contours(j+1)-1);
95 end
96 levelxe_abs = levelxe + startLocation(1) + vmax_tanker*t_step*cIter;
97 levelye_abs = levelye + startLocation(2);
98 plot(levelxe_abs,levelye_abs,linetypeTable{1},'color','k');
99 end
100 end
101
102 if ( nargin > 6 )
103 cOther = 1;
104 while (cOther<=length(gList))
105 grOther = gList{cOther};
106 if (size(grOther.N,1) > 2)
107 indexrOther = ceil((grOther.N(3)*(specific_theta+1))/2+1);
108 else
109 indexrOther = 1;
110 end
111 x1_grOther = linspace(grOther.min(1),grOther.max(1),grOther.N(1));
112 x2_grOther = linspace(grOther.min(2),grOther.max(2),grOther.N(2));
113 rsdataOther = rsData{floor((cOther+1)/2)}{end};
114 reachsetOther = ... 
115 contourc(x1_grOther,x2_grOther,rsdataOther(:, :, indexrOther)',[0,0]);
116 contoursOther=calcContours(reachsetOther);
117 for i=1:length(contoursOther)
118 if ( i == length(contoursOther) )
119 levelxOther=ref*reachsetOther(1,contoursOther(i)+1:end);
120 levelyOther=ref*reachsetOther(2,contoursOther(i)+1:end);
121 else
122 levelxOther=...
123
ref*reachsetOther(1, contoursOther(i)+1:contoursOther(i+1)-1);
levelyROther = ... 
ref*reachsetOther(2, contoursOther(i)+1:contoursOther(i+1)-1);
end 
levelxR_absOther = ...
levelxROther + startLocation(1) + vmax_tanker*t_step*cIter;
levelyR_absOther = ...
levelyROther + startLocation(2); 
plot(levelxR_absOther, levelyR_absOther, ... 
linetypeTable{floor((cOther+1)/2)},'color','b'); 
end 
cOther = cOther + 1;
if (nargin > 8 && size(usData,1)>0)
guOther = gList{cOther};
if (size(grOther.N,1) > 2)
  indexuOther = ceil(guOther.N(3)*(specific_theta+1)/2+1);
else
  indexuOther = 1;
end 
x1_guOther = linspace(guOther.min(1), guOther.max(1), guOther.N(1));
x2_guOther = linspace(guOther.min(2), guOther.max(2), guOther.N(2));
usdataOther = usData{floor(cOther/2)}{end};
reachsetOther=contourc(x1_guOther, x2_guOther, ....
usdataOther(:, :, indexuOther)', [0, 0]);
contoursOther=calcContours(reachsetOther);
for i=1:length(contoursOther)
  if (i == length(contoursOther))
    levelxUOther=ref*reachsetOther(1, contoursOther(i)+1:end);
    levelyUOther=ref*reachsetOther(2, contoursOther(i)+1:end);
  else
    levelxUOther=ref*reachsetOther(1, contoursOther(i)+1:contoursOther(i+1)-1);
    levelyUOther=ref*reachsetOther(2, contoursOther(i)+1:contoursOther(i+1)-1);
  end
  levelxU_absOther = levelxUOther + startLocation(1) +...
vmax_tanker*t_step*cIter;
  levelyU_absOther = levelyUOther + startLocation(2); 
  plot(levelxU_absOther, levelyU_absOther, ... 
  linetypeTable{floor(cOther/2)},'color','m'); 
end 
cOther = cOther + 1;
end 
else
cOther = 1;
while (cOther <= length(gList))
grOther = gList{cOther};
if (size(grOther.N,1) > 2)
    indexrOther = ceil(grOther.N(3)*(specific_theta+1)/2+1);
else
    indexrOther = 1;
end
x1_grOther = linspace(grOther.min(1),grOther.max(1),grOther.N(1));
x2_grOther = linspace(grOther.min(2),grOther.max(2),grOther.N(2));
rsdataOther = rsData{cOther}{end};
reachsetOther = contourc(x1_grOther,x2_grOther,...
    rsdataOther(:,1,indexrOther)',[0,0]);
levelxrOther = ref* reachsetOther(1,2:end);
levelyrOther = ref* reachsetOther(2,2:end);
levelx_absOther = levelxrOther + startLocation(1) + ...
    vmax_tanker*t_step*cIter;
levely_absOther = levelyrOther + startLocation(2);
plot(levelx_absOther,levely_absOther,...
    linetypeTable{cOther},'color','b');
cOther = cOther + 1;
end
x_3dof = cos(-locx3_prev)*uavx - sin(-locx3_prev)*uavy;
y_3dof = sin(-locx3_prev)*uavx + cos(-locx3_prev)*uavy;

huav = patch(x_3dof+startLocation(1)+...
    vmax_tanker*t_step*cIter+ref*waypoints(1,1),...
    y_3dof+startLocation(2)+ref*waypoints(1,2),uavc,'LineWidth',1.0);
htanker = patch(tx+startLocation(1)+vmax_tanker*t_step*cIter,...
    ty+startLocation(2),tc,'LineWidth',2.0);
plot(linspace(startLocation(1)-plotSize,...
    startLocation(1)+vmax_tanker*t_step*cIter,cIter+1),...
    zeros(1,cIter+1),'g');
plot(linspace(ref*locx1_prev+startLocation(1)-plotSize,...
    ref*locx1_prev+startLocation(1)+vmax_tanker*t_step*cIter,cIter+1,...
    ref*locx2_prev+ones(1,cIter+1),'r');
axis([plotCenter(1)-plotSize/2+startLocation(1)+vmax_tanker*t_step*cIter ...
    plotCenter(1)+plotSize/2+startLocation(1)+vmax_tanker*t_step*cIter ...]
    +plotCenter(2)+plotSize/2+startLocation(2)));
title(strcat('t=',numstr(t_step*cIter+inTime),'s'));
location = [startLocation(1) + vmax_tanker*t_step*cIter startLocation(2)];
outTime = inTime + t_step*(cIter - 1);
pause (0.1);
movieFrames (cIter) = getframe (gcf);
hold off;
cIter = cIter + 1;
if (get (handlesCopy . checkbox1 , 'Value')˜=0)
  uavData = shapeRectangleByCorners (esg ,... 
      [locx1_prev-4;locx2_prev-4;locx3_prev-pi/32;] ,... 
      [locx1_prev+4;locx2_prev+4;locx3_prev+pi/32;]);
  if (~isempty (find ((uavData<=0) & (esData{1}{end}<=0))))
    set (handlesCopy . pushbutton4 , 'Enable' , 'OFF');
  end
  if (~isempty (find ((uavData<=0) & (esData{2}{end}<=0))))
    set (handlesCopy . pushbutton5 , 'Enable' , 'OFF');
  end
  if (~isempty (find ((uavData<=0) & (esData{3}{end}<=0))))
    set (handlesCopy . pushbutton6 , 'Enable' , 'OFF');
  end
  if (~isempty (find ((uavData<=0) & (esData{4}{end}<=0))))
    set (handlesCopy . pushbutton7 , 'Enable' , 'OFF');
else
  if (~isempty (esData))
    set (handlesCopy . pushbutton4 , 'Enable' , 'ON');
    set (handlesCopy . pushbutton5 , 'Enable' , 'ON');
    set (handlesCopy . pushbutton6 , 'Enable' , 'ON');
    set (handlesCopy . pushbutton7 , 'Enable' , 'ON');
  else
    set (handlesCopy . pushbutton4 , 'Enable' , 'OFF');
    set (handlesCopy . pushbutton5 , 'Enable' , 'OFF');
    set (handlesCopy . pushbutton6 , 'Enable' , 'OFF');
    set (handlesCopy . pushbutton7 , 'Enable' , 'OFF');
end
end
while (get (handlesCopy . togglebutton1 , 'Value')==1)
  set (handlesCopy . togglebutton1 , 'String' , 'Continue');
pause (0.1);
end
set (handlesCopy . togglebutton1 , 'String' , 'Pause');
end

Listing B.5: displayEscape.m

function [location movieFrames] = displayEscape(tankPosition,...
escape, esg, esData, esTime, specific,theta)
global waypoints;
global handlesCopy;

plotCenter = [10 15];
plotSize = 200;
linetypeTable = {'-' '-.' '...';'};

escapeTime = 0.3;
cIter = length(esTime{escape});
ge = esg;
if(size(ge.N, 1) > 2)
    indexe = ceil(ge.N(3)*(specific_theta + 1)/2+1);
else
    indexe = 1;
end
x1_ge = linspace(ge.min(1), ge.max(1), ge.N(1));
x2_ge = linspace(ge.min(2), ge.max(2), ge.N(2));
locx1 = waypoints(2,1);
locx2 = waypoints(2,2);
locx3 = waypoints(2,3);
locx1_prev = waypoints(1,1);
locx2_prev = waypoints(1,2);
locx3_prev = waypoints(1,3);

parameterInit

[uavx, uavy, uavc] = uavpatch;
[tx, ty, tc] = tankerpatch;

scale=1;
uavx = scale*uavx;
uavy = scale*uavy;
tx = scale*tx;
ty = scale*ty;
tanker_x_offset = 15;

escape2ode
t_sim = 0;
c_sim = 1;
frameCount = 1;
startLocation = tankPosition;
x1 = X(:, 1);
x2 = X(:, 2);
x3 = X(:, 3);
ref = -1;
if ~exist('vmax_tanker')
    vmax_tanker = 0;
    ref = 1;
end
x_t = startLocation(1)*ones(size(t))+vmax_tanker*t;
y_t = zeros(size(t));
if ref == -1
    x_uav = x_t + ref*x1.*cos(ref*x3)+ ref*x2.*sin(ref*x3);
y_uav = y_t + ref*x1.*sin(ref*x3)+ ref*x2.*cos(ref*x3);
else
    x_uav = x_t + x1;
y_uav = y_t + x2;
end
while (cIter >=1)
esdata = esData{escape}{cIter};
escapeset = contourc(x1_ge, x2_ge, esdata(:, :, indexe)', [0, 0]);
contours = calcContours(escapeset);
while (t_sim <= esTime{escape}{end} - esTime{escape}{cIter} & c_sim <= length(t))
t_sim = t(c_sim);
    plot(ref*locx1_prev+startLocation(1)+vmax_tanker*t_sim , ... 
        ref*locx2_prev+startLocation(2) , 'ko');    hold on;
    for i = 1:length(contours)
        if ( i == length(contours) )
            levelxe = ref*escapeset(1, contours(i)+1:end);
        else
            levelxe = ref*escapeset(1, contours(i)+1:contours(i+1)-1);
        end
        levelye = ref*escapeset(2, contours(i)+1:contours(i)+1:contours(i+1)-1);
        levelxe_abs = levelxe + startLocation(1) + vmax_tanker*t_sim;
levy_e_abs = levy_e + startLocation(2);
plot(levy_e_abs,levy_e_abs,'-','color','r');
end

x_3dof = cos(ref*X(c_sim,3))*uavx - sin(ref*X(c_sim,3))*uavv;
y_3dof = sin(ref*X(c_sim,3))*uavx + cos(ref*X(c_sim,3))*uavv;

huav = patch(x_3dof+x_uav(c_sim),...
y_3dof+y_uav(c_sim),uavc,'LineWidth',1.0);
htanker = patch(tx+x_t(c_sim),ty+y_t(c_sim),tc,'LineWidth',2.0);
plot(x_uav(1:c_sim),y_uav(1:c_sim),'r');
plot(x_t(1:c_sim),y_t(1:c_sim),'g');
axis([plotCenter(1)-plotSize/2+startLocation(1)+vmax_tanker*t_sim ...
    plotCenter(1)+plotSize/2+startLocation(1)+vmax_tanker*t_sim ...
    plotCenter(2)-plotSize/2+startLocation(2) ...
    plotCenter(2)+plotSize/2+startLocation(2)]);
hold off;
title(strcat('t=';num2str(t_sim),'s'));
location = [startLocation(1) + vmax_tanker*t_sim startLocation(2)];
waypoints = [ref*X(c_sim)+x_t(c_sim) ref*y_uav(c_sim)+y_t(c_sim) ...
    ref*X(c_sim,3);waypoints(2,:)];

pause(0.2);
c_sim = c_sim + 1;
movieFrames(frameCount) = getframe(gcf);
frameCount = frameCount + 1;
end

cIter = cIter - 1;

while(get(handlesCopy.togglebutton1,'Value')==1)
    set(handlesCopy.togglebutton1,'String','Continue');
pause(0.1);
end
set(handlesCopy.togglebutton1,'String','Pause');
end

Listing B.6: displayTransition.m

function [location movieFrames] = displayTransition(tankPosition,...
    maneuver,gList,rsData,rsTime,...
    usData,usTime,specific_theta)
% list of inputs:
% tankPosition is the initial location of tank
% waypoints is a 2 by 3 matrix for current location and previous location:
% maneuver depends on the transition ode function;
% rsData, rsTime, usData and usTime are the data inputs;
% specific_theta is the contour slice for display;

global waypoints;
global handlesCopy;

if (nargin < 8)
    specific_theta = 0;
end

if (nargin > 5 && size(usData,1)>0)
    gu = gList{2};
    if (size(gu.N,1) > 2)
        indexu = ceil(gu.N(3)*(specific_theta+1)/2+1);
    else
        indexu = 1;
    end
    x1.gu = linspace(gu.min(1),gu.max(1),gu.N(1));
    x2.gu = linspace(gu.min(2),gu.max(2),gu.N(2));
end

plotCenter = [0 0];
plotSize = 100;
gr = gList{1};
cIter = length(rsTime{1});
if (size(gr.N, 1) > 2)
    indexr = ceil(gr.N(3)*(specific_theta+1)/2+1);
else
    indexr = 1;
end
x1.gr = linspace(gr.min(1),gr.max(1),gr.N(1));
x2.gr = linspace(gr.min(2),gr.max(2),gr.N(2));
locx1 = waypoints(2,1);
locx2 = waypoints(2,2);
locx3 = waypoints(2,3);
locx1.prev = waypoints(1,1);
locx2.prev = waypoints(1,2);
locx3.prev = waypoints(1,3);

parameterInit
\[ [uavx, uavy, uavc] = uavpatch; \]
\[ [tx, ty, tc] = tankerpatch; \]
\[ scale = 1; \]
\[ uavx = scale * uavx; \]
\[ uavy = scale * uavy; \]
\[ tx = scale * tx; \]
\[ ty = scale * ty; \]
\[ tanker_x_offset = 15; \]
\[ tx = tx + tanker_x_offset; \]
\[ maneuver2ode \]
\[ t_sim = 0; \]
\[ c_sim = 1; \]
\[ startLocation = tankPosition; \]
\[ frameCount = 1; \]
\[ x1 = X(:, 1); \]
\[ x2 = X(:, 2); \]
\[ x3 = X(:, 3); \]
\[ ref = -1; \]
\[ if ('exist ('vmax_tanker')') \]
\[ vmax_tanker = 0; \]
\[ ref = 1; \]
\[ end \]
\[ x_t = startLocation(1)*ones(size(t))+vmax_tanker*t; \]
\[ y_t = zeros(size(t)); \]
\[ if ref == -1 \]
\[ x_uav = x_t + ref*x1.*cos(ref*x3)+ ref*x2.*sin(ref*x3); \]
\[ y_uav = y_t + ref*x1.*sin(ref*x3)+ ref*x2.*cos(ref*x3); \]
\[ else \]
\[ x_uav = x_t + x1; \]
\[ y_uav = y_t + x2; \]
\[ end \]
\[ while (cIter >= 1) \]
\[ rsdata = rsData{1}{cIter}; \]
\[ reachsetr = contourc(x1_gr, x2_gr, rsdata(:, :, indexr)', [0, 0]); \]
\[ contoursr = calcContours(reachsetr); \]
\[ % levelxr = reachset(1, 2:end); \]
\[ % levelyr = reachset(2, 2:end); \]
if (nargin>5 & size(usData,1)>0)
    if (cIter<=length(usTime{1}))
        usdata = usData{1}{cIter};
    else
        usdata = usData{1}{end};
    end
    reachsetu = contourc(x1_gu,x2_gu,usdata(:,indexu)',[0,0]);
    contoursu = calcContours(reachsetu);
    levelxu=reachset(1,2:end);
    % levely=reachset(2,2:end);
    end
while (t_sim <= rsTime{end}-rsTime{1}{cIter} && c_sim <= length(t))
    plot(ref*locx1_prev+startLocation(1)+vmax_tanker*t_sim,...
         ref*locx2_prev+startLocation(2), 'ko'); hold on;
    plot(ref*locx1+startLocation(1)+vmax_tanker*t_sim,...
         ref*locx2+startLocation(2), 'ko');
    for i=1:length(contoursr)
        if ( i == length(contoursr) )
            levelx = ref*reachsetr(1,contoursr(i)+1:end);
            levely = ref*reachsetr(2,contoursr(i)+1:end);
        else
            levelx = ref*reachsetr(1,contoursr(i)+1:contoursr(i+1)-1);
            levely = ref*reachsetr(2,contoursr(i)+1:contoursr(i+1)-1);
        end
        levelx_abs = levelx + startLocation(1) + vmax_tanker*t_sim;
        levely_abs = levely + startLocation(2);
        plot(levelx_abs,levely_abs,'-','color','b');
    end
end
if (nargin>5 & size(usData,1)>0)
    for i=1:length(contoursu)
        if ( i == length(contoursu) )
            levelx = ref*reachsetu(1,contoursu(i)+1:end);
            levely = ref*reachsetu(2,contoursu(i)+1:end);
        else
            levelx = ref*reachsetu(1,contoursu(i)+1:contoursu(i+1)-1);
            levely = ref*reachsetu(2,contoursu(i)+1:contoursu(i+1)-1);
        end
        levelx_abs = levelx + startLocation(1) + vmax_tanker*t_sim;
        levely_abs = levely + startLocation(2);
        plot(levelx_abs,levely_abs,'-','color','m');
    end
end

x_{3dof} = \cos(\text{ref} \times X(c_{\text{sim}}, 3)) \times uavx - \sin(\text{ref} \times X(c_{\text{sim}}, 3)) \times uavy;

y_{3dof} = \sin(\text{ref} \times X(c_{\text{sim}}, 3)) \times uavx + \cos(\text{ref} \times X(c_{\text{sim}}, 3)) \times uavy;

huav = \text{patch}(x_{3dof} + x_{uav}(c_{\text{sim}}), ... 
    y_{3dof} + y_{uav}(c_{\text{sim}}), uavc, \text{\'LineWidth\',} 1.0);

htanker = \text{patch}(tx+x_t(c_{\text{sim}}), ty+y_t(c_{\text{sim}}), tc, \text{\'LineWidth\',} 2.0);

plot(x_{uav}(1:c_{\text{sim}}), y_{uav}(1:c_{\text{sim}}), \text{\'r\'});

plot(x_t(1:c_{\text{sim}}), y_t(1:c_{\text{sim}}), \text{\'g\'});

axis([\text{plotCenter}(1) - \text{plotSize}/2 + \text{startLocation}(1) + \text{vmax}_{\text{tanker}} \times t_{\text{sim}} ... 
    \text{plotCenter}(1) + \text{plotSize}/2 + \text{startLocation}(1) + \text{vmax}_{\text{tanker}} \times t_{\text{sim}} ... 
    \text{plotCenter}(2) - \text{plotSize}/2 + \text{startLocation}(2) ... 
    \text{plotCenter}(2) + \text{plotSize}/2 + \text{startLocation}(2)])

hold off;

title(strcat('t = ', num2str(t_{\text{sim}}), \text{\'s\'}));

location = [\text{startLocation}(1) + \text{vmax}_{\text{tanker}} \times t_{\text{sim}} \text{ startLocation}(2)];

waypoints = [\text{ref} \times x_{uav}(c_{\text{sim}}) + x_{t}(c_{\text{sim}}) \text{ ref} \times y_{uav}(c_{\text{sim}}) + y_{t}(c_{\text{sim}}) ... 
    \text{ref} \times X(c_{\text{sim}}, 3); waypoints(2,:)]

pause(0.2);

\text{t}_{\text{sim}} = \text{t}(c_{\text{sim}});

\text{c}_{\text{sim}} = \text{c}_{\text{sim}} + 1;

\text{movieFrames(frameCount)} = \text{getframe(gcf)};

\text{frameCount} = \text{frameCount} + 1;

end

c_{\text{Iter}} = c_{\text{Iter}} - 1;

while (get(handlesCopy.togglebutton1, \text{\'Value\'}) == 1)

    \text{set(handlesCopy.togglebutton1, \text{\'String\'}, \text{\'Continue\'})};

    pause(0.1);

end

\text{set(handlesCopy.togglebutton1, \text{\'String\'}, \text{\'Pause\'})};

d{end}
REFERENCES


