Southern Illinois University Carbondale
OpenSIUC

Theses | Theses and Dissertations

5-1-2020

PARALLELIZED ROBUSTNESS COMPUTATION FOR CYBER PHYSICALSYSTEMS VERIFICATION

Joseph Cralley
Southern Illinois University Carbondale, jkolecr@gmail.com

Follow this and additional works at: https://opensiuc.lib.siu.edu/theses

Recommended Citation
Cralley, Joseph, "PARALLELIZED ROBUSTNESS COMPUTATION FOR CYBER PHYSICALSYSTEMS VERIFICATION" (2020). *Theses*. 2668.
https://opensiuc.lib.siu.edu/theses/2668

This Open Access Thesis is brought to you for free and open access by the Theses and Dissertations at OpenSIUC. It has been accepted for inclusion in Theses by an authorized administrator of OpenSIUC. For more information, please contact opensiuc@lib.siu.edu.

PARALLELIZED ROBUSTNESS COMPUTATION FOR CYBER PHYSICAL SYSTEMS VERIFICATION

by

Joseph Cralley

B.S., Southern Illinois University Carbondale, 2018

A Thesis
Submitted in Partial Fulfillment of the Requirements for the
Master of Science Degree

School of Computing
in the Graduate School
Southern Illinois University Carbondale
May 2020

THESIS APPROVAL

PARALLELIZED ROBUSTNESS COMPUTATION FOR CYBER PHYSICAL SYSTEMS VERIFICATION

by

Joseph Cralley

A Thesis Submitted in Partial

Fulfillment of the Requirements

for the Degree of

Master of Science

in the field of Computer Science

Approved by:

Dr. Henry Hexmoor, Chair

Dr. Bardh Hoxha, Co-Chair

Dr. Banafsheh Rekabdar

Dr. Norman Carver

Graduate School
Southern Illinois University Carbondale
April 1, 2020

AN ABSTRACT OF THE THESIS OF

Joseph Cralley, for the Master of Science degree in Computer Science, presented on April 01, 2020, at Southern Illinois University Carbondale.

TITLE:PARALLELIZED ROBUSTNESS COMPUTATION FOR CYBER PHYSICAL SYSTEMS VERIFICATION

MAJOR PROFESSOR: Dr. Henry Hexmoor

Failures in cyber physical systems can be costly in terms of money and lives. The mars climate orbiter alone had a mission cost of 327.6 million USD which was almost completely wasted do to an uncaught design flaw. This shows the importance of being able to define formal requirements as well as being able to test the design against these requirements. One way to define requirements is in Metric Temporal Logic (MTL), which allows for constraints that also have a time component. MTL can also have a distance metric defined that allows for the calculation of how close the MTL constraint is to being falsified. This is termed robustness.

Being able to calculate MTL robustness quickly can help reduce development time and costs for a cyber physical system. In this thesis, improvements to the current method of computing MTL robustness are proposed. These improvements lower the time complexity, allows parallel processing to be used, and lowers the memory foot print for MTL robustness calculation. These improvements will hopefully increase the likelihood of MTL robustness being used in systems that were previously inaccessible do to time constraints, data resolution or real time systems that need results quickly. These improvements will also open the possibility of using MTL in systems that operate for a large amount of time and produce a large amount of signal data.

ACKNOWLEDGMENTS

I would like to thank:

Dr. Bardh Hoxha

Dr. Henry Hexmoor

Ourania Spantidi

DEDICATION

I dedicate this paper to: To my loving parents Jean and Kevin who have supported me all these years. They have always been there for me when I have needed them most.

To my fiance Megan who has given me strength and motivation over the years. Her love has pushed me to work hard and always try my best.

To my colleague Ourania who has been a mentor and a friend to me. I could not imagine making it this far without her help.

TABLE OF CONTENTS

CHAPTER PAGE

ABSTRACT i

ACKNOWLEDGMENTS ii

DEDICATION iii

LIST OF TABLES vi

LIST OF FIGURES vii

LIST OF ALGORITHMS viii

CHAPTERS

CHAPTER 1 - Introduction 1

1.1 Contribution 4

CHAPTER 2 - Related Work 5

CHAPTER 3 - Preliminaries 7

3.1 MTL 7

CHAPTER 4 - TLTk overview 10

4.1 Structure 10

4.1.1 Signals and System 10

4.1.2 Calculation Structure 11

4.1.3 Predicate Calculation 12

4.1.4 Logical MTL operations 13

4.1.5 Temporal MTL operations 14

CHAPTER 5 - Experimental Results 20

5.1 Hardware and Software 20

5.2 Serial Execution vs Parallel Execution 20
5.3 TLTk vs Breach 21
5.4 TLTK vs STALIRO 23
CHAPTER 6 - Conclusion and Future Work 25
REFERENCES 26
VITA 28

LIST OF TABLES

TABLE		PAGE
Table 3.1	MTL logical semantics	8
Table 3.2	MTL robustness semantics	9
Table 5.1	Threaded vs Unthreaded TLTk time in seconds	21
Table 5.2	One dimensional formula list	21
Table 5.3	Larger Dimensional Formulas	24

LIST OF FIGURES

FIGURE PAGE

Figure 4.1 Parse Tree of MTL formula $\varphi = \Diamond(\neg r1 \wedge r2)$ 11

Figure 4.2 Memory table of $\varphi = \Diamond(\neg r1)$ 12

Figure 5.1 One dimensional execution time(seconds) 22

Figure 5.2 Higher dimensional execution time(seconds) 23

LIST OF ALGORITHMS

ALGORITHM PAGE

Algorithm 1 Not sub task. 13

Algorithm 2 And sub task. 14

Algorithm 3 Or sub task. 14

Algorithm 4 find_max($robustness, start, end$) 16

Algorithm 5 find_min($robustness, start, end$) 16

Algorithm 6 search_sorted($\tau, start, end, time$) 17

Algorithm 7 FINALLY(I,ε,τ,ς) 18

Algorithm 8 GLOBAL(I,ε,τ,ς) 19

CHAPTER 1

INTRODUCTION

Cyber physical systems (cps) is a mechanism where an algorithm has to interact with or control some piece of physical hardware. Modern cps are becoming more complex and integral to everyday life for an increasing amount of people. It has even become commonplace for people to trust their lives to the performance of a cps. For instance, most newly produced cars do not have a direct physical connection between the controls and the wheels and motor. The signals from the controls are sent through the car's computer, which processes them and then sends a signal to the motor controls and steering controls. This makes it very important to test the software that those computers run rigorously. When rigorous testing is not performed on the algorithms errors can easily slip into the finished product. One example is the Mars Climate Orbiter(MCO) failure.[5] MCO was launched on top of a Delta II launch vehicle on December 11, 1998 to profile the structure of the martian atmosphere. It failed its Mars Orbit Insertion maneuver, which caused loss of the signal and a mission failure. The root cause listed for the mission failure was noted as "Failure to use metric units in the coding of a ground software file, 'SmallForces', used in trajectory models" with a primary contributing factor of "Undetected mismodeling of spacecraft velocity changes". This incident shows the importance of having formal Model Checking Techniques[9].

Model Checking has become integral to modern engineering techniques. A model is an easier to produce representation of an engineering project. A model can be a physical small scale representation of the project or a mathematical representation of the project. This paper will mostly focus on mathematical models that can be simulated on a computer. Models are often cheaper to make than implementing a full scale project and can find design flaws that make projects both easier to implement and more efficient. These models are designed by engineers that make certain assumptions about what the intended system will perform under. These assumptions can be wrong, as seen in the MCO launch, where

they assumed the wrong system of measurements. After the model is made it is tested under simulated conditions, which will produce an output signal of the states that the model reaches at different times in the simulation. The state signal that the simulation produces of the model can be used to check if the model performed successfully or if it reached some sort of bad state. Before running the simulation, the starting condition of the model is defined. If a car is used as an example this could be variables like velocity, position, and angle. There may be multiple starting configurations in the car example, such as the velocity being between 20 and 30 mile per hour. All the possible start conditions of a model about to be simulated are labeled the starting state space. From the starting state space there are reachable states and possibly states that can not be reached. An example of an unreachable state is you being unable to drive a car faster than its max speed. Any state with a speed faster than that car's max is unreachable.

When designing a model it is important to define a desired state pace and a undesired state space. These state spaces define what the designer wants the model to do and what it should not do. Looking at the MCO, its desired state space was a specific orbit around mars. However, due to the wrong units of measurement being used it reached the undesirable state space of being not in an orbit around mars. This means the velocity and position of the satellite do not exist inside the set of desired states. One of the main questions that can be asked is the possibility for a cps to reach a undesirable state. Someone's initial attempt to answer this question might be to successfully search all possible states of the model. This can be very effective if the feasible state space is a small finite space. Unfortunately, this is not the case for most models because most models have infinite state spaces. For example, the MCO would want to model MCO's velocity. Velocity is a continuous variable, so any range of velocities where the lower bound and upper bound are not equal has infinite points to search.

One commonly used method to deal with infinite state spaces is theorem proving. Theorem proving involves abstracting the the cps down to important mathematical

properties. By posing the problem as theorems and lemmas it is possible to prove if a specification is satisfied or not. These methods can be very useful in some situations since they cover can cover the entire state space. However, they are often very complex and work intensive to set up properly. This task can be made easier by using automated theorem proving techniques.

When running simulations of models, the time the model exists in a certain state space could be important. For example, a processor might be able to have a current temperature hotter than its max safe temperature. As long as it returns back to safe temperatures within a specific time frame there should be no damage. A language that allows for the expression of such temporal relationships is metric temporal logic (MTL).[6] MTL allows for temporal statements to be applied to boolean predicates. An example of a statement that can be made with MTL could be Finally($\Diamond$) MCO must reach a stable orbit around mars. This means there is some point in time where MCO reaches a state that it is in a stable orbit. There are also the global($\Box$) and until(U) expressions. Global means the predicate has to always be true for the global statement to be true. Until has two operands and says the left operand must be true until the right operand becomes true. These formulas can be nested as well. For example, let us say $r1$ is that MCO is in a stable orbit around mars. Then the following MTL formula $\Diamond(\Box(r1))$ means there is some point in MCO's mission that it will always stay in a stable orbit. This type of expressiveness can make MTL very powerful for defining formulas that models have to adhere to in order to be considered correct.

Having MTL formulas just returning true or false is useful, but they can also be extended to return more relevant information. [4] Predicates can be changed to be bounded sets that have some sort of distance metric. The distance metric for the set defined by a predicate should be positive when outside the set and negative when inside the set. This distance can be used in MTL formulas to produce a robustness value that indicate how far away an MTL formula is from switching states. This robustness can help

engineers by providing a metric to determine how close a model is from failing. The robustness can also be used in an optimization algorithm like simulated annealing to help find maximums or minimums in robustness value.[10] Engineers might want to search for minimums in robustness value to find points where there models fail.

There are a few tools that compute MTL robustness, but this paper will focus on two of them. The two tools looked at in this paper will staliro and breach.[13][11][12]. The contributions of this paper will be based on staliro's temporal robustness calculation algorithm dp_taliro.[1]

1.1 CONTRIBUTION

In this paper, a change to the dp_taliro is suggested that allows for a reduction of time complexity and opens the possibility of parallel computing to be used when computing MTL robustness. The first change is made to the order in which the algorithm processes time steps. Processing all the values in one subformula before processing the next instead of processing all subformulas on a time step opens up the possibility for parallel processing. The second change is made to the way lower and upper bound time steps are found. The dp_taliro algorithm did extra searching through time steps that turns out to be unnecessary. This change lowers the time complexity based on the interval size. Finally a change to how memory is managed is suggested to allow for a smaller memory usage. This change can drastically impact how much memory is needed at one time at the cost of needed to allocate and deallocate memory when needed.

CHAPTER 2

RELATED WORK

One of the first implementations of temporal logic is Linear Temporal Logic (LTL) [7]. LTL logic allows for a definition of a temporal formula with a few base operators. It then takes in a string and see if that string matches what the formula describes. If the string is with in the formula's parameters then the statement is true otherwise it is false. When processing LTL the process string gets passed to predicates. The process string gets read from the lowest time step to highest and the predicate produces a new string with characters of true or false.

LTL has been extensively used for verification. In the paper [8] LTL logic is used to verify modeled circuits. The modeling software used is SystemC which is a C++ library that allows for event based modeling and simulations. Since the they want to check digital circuits they are able to perform proof by exhaustion verification techniques on the circuits. This checks all the states of the circuit to see wither if violates the LTL formula or not. This application is just one example of the usefulness of LTL logic however it does have its limitations.

LTL logic can only define constraints over the entire input string. This can be very useful but there are some applications where being able to look over a certain time interval is crucial. Metric Temporal Logic (MTL) adds the necessary syntax and semantic to LTL to facilitate this. The MTL syntax adds time intervals to LTL operations which specify how far in the future does this specification holds. MTL has been expanded further in [4] to move from boolean statements to statements that produce a distance metric. This distance indicates how far the formula is to being falsified and is called MTL robustness. This robustness metric has been used in many applications in cyber physical systems. In the paper [3] the authors use MTL robustness to verify the path finding of drones. To do this they use MTL robustness to define safety margins for an enhanced A* algorithm.

MTL robustness calculation is introduced in [4] and has been implemented in a few

algorithms. One of the early implementations was in MATLAB with the FW-TALIRO(ForWard algorithm for TemporAl LogIc RObustness [13]. It was part of the S-TALIRO tool box for MATLAB. The FW-TALIRO algorithm uses formula rewriting techniques to calculate robustness. This algorithm has a high time complexity and a faster approach was developed. S-TALIRO changed its robustness calculation algorithm to DP-TALIRO [1], which used a table-based approach to calculate robustness. This table is computed with one algorithm that uses if statements to decide what operation to use. This can make the implementation compact but also makes it harder for others to add to the system. The S-TALIRO implementation is very flushed out; it is able to calculate the robustness of higher dimensional predicates in several forms and has been highly tested. The S-TALIRO version of DP-TALIRO also has a parser that can take in MTL formulas as strings of text. However, the S-TALIRO implementation of DP-TALIRO exaction time increases very rapidly when a signal has a large amount of time steps or the MTL formula is very large. S-TALIRO with the DP-TALIRO algorithm has been tested in automotive applications as seen in [16].

Breach is another implementation of a MTL robustness calculation tool [11] It is also designed to work in MATLAB. It has the advantage of not having the large increase of execution time over staliro. However it is hard to compare staliro and breach directly since breach seems to not support higher dimensional predicates. This means that if two or more variables depend on each other breach will not be able to model that. Breach works on a substask structure where each MTL operation is treated independent of each other. This means it should be easy to add new operations to breach.[15]

CHAPTER 3

PRELIMINARIES

3.1 MTL

MTL is an extension of boolean logic that allows for statments through intervals of time. MTL is a recursive syntax that consists of

$$\varphi \models true|predicate|\varphi_1 \wedge \varphi_2|\neg\varphi_1|\varphi_1 U_I \varphi_2|X\varphi_1$$

A predicate is an atomic proposition that can either be true or false. $\wedge, \neg$ are unchanged from normal logical $\wedge, \neg$. The U_I is the until operator where I is and interval $[a, b] : a, b \in \mathbb{R}^+ : 0 \leq a < b$. The X is the next operator. Most implementations of MTL also include logical operators $\implies, \vee, \Leftrightarrow$ and temporal operators $\Diamond_I\varphi|\Box_I\varphi$. $\Diamond_I$ stands for the finally and $\Box_I$ stands for global operations. However, these can be derived from the base MTL operations provided as such:

$$\Diamond_I\varphi = trueU_I\varphi$$

$$\Box_I\varphi = \neg\Diamond_I\neg\varphi$$

The input to MTL formulas are a signal (s) that can come from a simulation or a cps. This signal (s) consists of two parts: σ, which is the set of states, and τ, which is the set of time steps. τ and σ are the same cardinality, meaning there is a bijection between the sets. Let $i \in \mathbb{N}^+$ where $i < |\tau|$ then the state $\sigma[i]$ occurred at time $\tau[i]$. The set τ is monotonic; it increases in value as i increases or $\tau[i] < \tau[i+1]$. With s $= (\tau, \sigma)$ the definition of the semantics at time step i can be given as seen in table 3.1.

This is the semantics for boolean MTL; however, it can be expanded to produce useful information on how far away the formula is from being True or False. This is known as MTL robustness. The first major change to the Boolean MTL semantics is that predicates

Table 3.1: MTL logical semantics

$(s,i) \models true$	MTL formula is true
$(s,i) \models false$	MTL formula is false
$(s,i) \models predicate$	True if the predicate is true at time step i given signal s else false
$(s,i) \models \neg\varphi$	True if φ is False at time step i else False
$(s,i) \models \varphi_1 \wedge \varphi_2$	True if φ_1 and φ_2 is True at time step i else False
$(s,i) \models \varphi_2 \vee \varphi_2$	False if φ_1 and φ_2 is False at time step i else True
$(s,i) \models \Box_I\varphi$	True if φ is True for all $i \in I$ else False
$(s,i) \models \Diamond_I\varphi$	False if φ is False for all $i \in I$ else True
$(s,i) \models \varphi_1 U \varphi_2$	True if there exists $j \leq i$ while φ_2 at time step j is True with $j \in i + I$ and for all $j > k > i$ where φ_1 at time step k is True

have to be changed to sets. These sets need to have a defined boundary that has a defined distance metric. If the trace from the signal is out of the predicate set the robustness will be the shortest distance to the boundary of the set. If the signal is a point in the predicate set the robustness is the shortest distance to the closest point on the boundary of the set times negative one. This distance to the boundary in the set is normally called depth. Let X be a set with a boundary that has a distance metric defining a predicate:

$$predicate\ robustness(\sigma, X) = \begin{cases} distance(\sigma_i, X) & if \sigma_i \in X \\ -depth(\sigma_i, X) & \sigma_i \notin X \end{cases}$$

The signed distance calculated with the predicate robustness function is then manipulated by the other MTL operations in the formula. If the formula is $\Diamond_I(pred)$, pred is calculated first before the Finally. The semantics for MTL robustness can be seen in table 3.2. The operations $\Diamond_I, \Box_I$, and U_I need to work on a certain subset of τ. This subset has a lower bound time step of $\tau[i + I[0]]$ and a upper bound of $\tau[i + I[1]]$. If $i + I[i]$ is greater than $|\tau|$ the largest index in τ is used. In these operators, the defined subset is searched to produce the robustness for that operator and will be labeled τ'.

The order in which these semantics are applied are different between the dp_taliro [1]

Table 3.2: MTL robustness semantics

$(s,i) \models true$	MTL formula is ∞
$(s,i) \models false$	MTL formula is $-\infty$
$(s,i) \models predicate$	$predicate_robustness(s_i, X)$
$(s,i) \models \neg\varphi$	$-\varphi(s,i)$
$(s,i) \models \varphi_1 \wedge \varphi_2$	$\varphi_1(s,i) \sqcap \varphi_2(s,i)$
$(s,i) \models \varphi_2 \vee \varphi_2$	$\varphi_1(s,i) \sqcup \varphi_2(s,i)$
$(s,i) \models \Box_I\varphi$	$\sqcap_{j\in\tau'}\varphi(s_j, j)$
$(s,i) \models \Diamond_I\varphi$	$\sqcup_{j\in\tau'}\varphi(s_j, j)$
$(s,i) \models \varphi_1 U \varphi_2$	

and what this paper proposes. In the dp_taliro algorithm, they compute the all semantics on that time step at once. This paper proposes to apply the semantic to all time steps before moving up to the next semantic. For instance, if dp_taliro was to calculate $\Box_I(\varphi)$ it would calculate time step $|\tau|$ for φ and then time step $|\tau|$ for $\Box$. After computing those two values dp_taliro will move on to computing the two values for $|\tau| - 1$ and will continue computing two values at a time till it finishes at time step zero. More details about calculation order can be seen in chapter three.

CHAPTER 4

TLTK OVERVIEW

4.1 STRUCTURE

TLTk is a python and C-based implementation of the divide and conquer algorithm to compute MTL robustness. The main user interface is python, which allows the end user to not have to think about memory management or parallel processing. Cython is used to allow C and Python to talk to each other. Cython translates the python objects into C data types. Cython is also used to talk to the CUDA library for robustness calculation on the GPU. TLTk is passed a signal set and a set of time stamps for when those signals are observed. These signals can be generated from a simulation or gathered from sensors.

4.1.1 Signals and System

When using TLTk for robustness calculation, two python objects need to be created by the user:

- traces: {Python dictionary}

  This is a python dictionary that is passed to TLTk at execution time. The keys of the dictionary are strings containing the name of the signal. The data stored at the key index is a python list. If the signal is one-dimensional then it is stored in a one-dimensional python list list of $\|\tau\|$ length. If the signal has a dimension greater than one it is stored in a two-dimensional python list.

- timestamps: {Python list}

  This python list represents the set τ and has length $\|\tau\|$. The value at timestamps[i] represents the time at which traces['key'][i] was taken. TLTk assumes that all data sets were sampled synchronously, so only one list of timestamps are needed. The time stamps have to be stored in increasing order to get accurate robustness calculations.

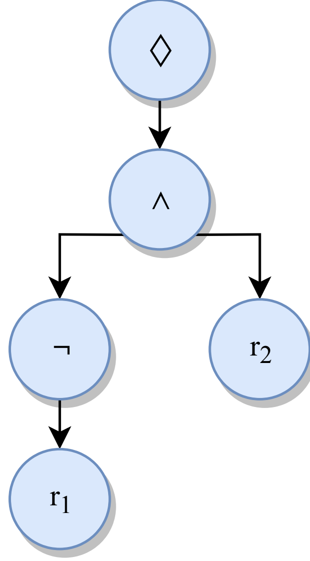


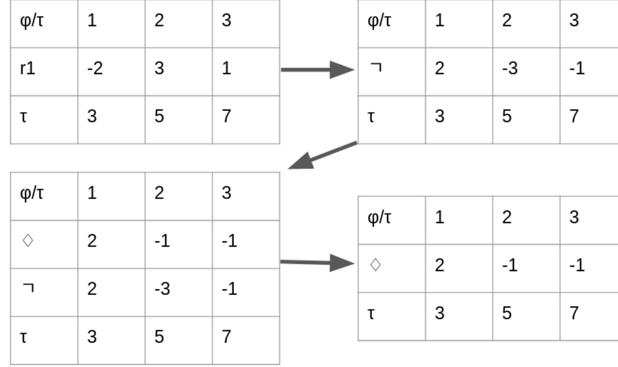
Figure 4.1: Parse Tree of MTL formula $\varphi = \Diamond(\neg r1 \wedge r2)$

4.1.2 Calculation Structure

When TLTk processes MTL formulas it processes it in the structure of a tree. It is up to the creator of the MTL operation if they give priority to left branch or right branch. Currently, all MTL operations give priority to the left branch. Let us look at the calculation of the MTL formula $\varphi = \Diamond(\neg r1 \wedge r2)$. Figure 4.1 shows how φ breaks down in to the parse tree.

The first node in the tree to get processed is the leftmost leaf node, which in this case is **r1**. The results are then returned to the $\neg$ operation, which is then processed. The results from both sub branches are needed for the $\wedge$ operation because it has two operands. That means **r2** is processed before the $\wedge$ operation. After **r1** and $\wedge$ have been processed the $\Diamond$ operation calculates its robustness.

The memory usage can be represented as a table with the rows being the memory each sub formula uses and each column being the memory that sub formula uses at that time step. That means there are $|\tau|$ columns and at most $|\varphi|$ rows. In the DP_taliro algorithm, the table that stores all the values gets allocated at the start of execution. This means the memory requirements for MTL formulas will be higher than is needed. MTL



φ/τ	1	2	3
r1	-2	3	1
τ	3	5	7

φ/τ	1	2	3
¬	2	-3	-1
τ	3	5	7

φ/τ	1	2	3
◊	2	-1	-1
¬	2	-3	-1
τ	3	5	7

φ/τ	1	2	3
◊	2	-1	-1
τ	3	5	7

Figure 4.2: Memory table of $\varphi = \Diamond(\neg r1)$

operations like $\wedge, \vee, \neg$ can be calculated in place without allocating memory specifically for the operations. As a value is calculated it can be stored in the column being calculated. Operations like $\Diamond_I, \Box_I$, and U_I need to allocate a row when performed because the results need to be stored somewhere without changing any of the data stored in the sub formulas. Figure 4.2 is an example of memory usage when calculating the MTL formula $\Diamond(\neg r1)$. The first table is the memory usage of calculating the predicate, which requires a row for the time stamps $\boldsymbol{\tau}$ and a row for the predicate values. The second table shows the $\neg$ operation overwriting the predicate values. The third table represents a row being allocated so the results of the $\Diamond$ operation can be stored. The row containing the $\neg$ is deleted because it will not be needed again.

4.1.3 Predicate Calculation

TLTk uses a modified version of the DP_taliro algorithm for robustness calculation. TLTk breaks the DP_taliro algorithm into sub tasks, which then divides the work in between threads. This allows for a significant speed up in calculation when compared to the original DP_taliro algorithm. The first computation that takes advantage of parallel processing is the predicate distance metric calculation. This calculation finds the distance of the current time step from the set

$$Ax \leq b$$

Distance is calculated by solving the optimization problem. When a node is processed, all time steps for that node are calculated before it moves on to the next. This is different than the DP_taliro algorithm, where it calculates all sub formulas on a time step before it moves on.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & ||x - s|| \\ \text{subject to} \quad & Ax \leq b, \end{aligned}$$

This minimization problem is reworked into the quadratic programming problem

With **s** being a n by 1 matrix representing the value of the signal at the current time step being calculated. The n by 1 vector **x** is the optimizer. When the problem is solved **x** will be the closet point in the set $Ax \leq b$ to the signal point **s** at that time step. TLTk uses the Goldfarb/Idnani algorithm to solve the above quadratic programming problem.[14] Each time steps distance is independent of each other. This means that all of time steps can be calculated in parallel and not sequentially as seen in the DP_taliro algorithm.

4.1.4 Logical MTL operations

The logical operators are $\wedge, \vee$, and $\neg$. When calculating these logical operators, only one time step is looked at a time. This allows for the possibility of dividing the calculation of all the time steps into threads. However, with most of the logical operations being very low time complexity operations to be preformed on each time step, that might not be worth the thread start up cost. Processing the time steps in logical operators in parallel will only be time efficient with large trace sizes. This is so the threads can pay back the cost of starting them. The $\neg$ operation is depicted in algorithm 1.

Algorithm 1: Not sub task.

```
1 Not (¬) subtask
  Data: Robustness array not_robustness
2 for i ← |τ| to 0 do
3   not_robustness[i] ← −1 ∗ not_robustness[i]
```

The ¬ takes in an array **not_robustness**, which is the robustness calculated from the sub formula of the not operation. It is also used to store the results of the ¬ operation in place. In practice, **not_robustness** would be an address to the memory in which the sub formula robustness in stored.

The ∧ and ∨ operations are very similar to each other, with the only difference being that ∧ takes the min and ∨ takes the max of the current time steps. The subtask algorithms for ∧ and ∨ can be seen in algorithm 2 and algorithm 3, respectively.

Algorithm 2: And sub task.

```
1 And (∧) subtask
  Data: Robustness array left_robustness
  Data: Robustness array right_robustness
2 for i ← |τ| to 0 do
3   left_robustness[i] ← min(left_robustness[i], right_robustness)
```

Algorithm 3: Or sub task.

```
1 Or (∨) subtask
  Data: Robustness array left_robustness
  Data: Robustness array right_robustness
2 for i ← |τ| to 0 do
3   left_robustness[i] ← max(left_robustness[i], right_robustness)
```

4.1.5 Temporal MTL operations

The temporal operators are $\Diamond, \Box$, and U. To compute one time step of these operations, other time steps must be looked at. That means the input data can not be changed, otherwise it would produce the wrong results. When running these operations memory specifically for the results of the operations need to be allocated.

Temporal operations algorithms and optimizations

The temporal operators have an optimization for when the formula have bounds of $I = [0, \infty]$. When $I = [0, \infty]$ it is possible to only look at one time or two time steps at a

time. This means τ and robustness will not need to be searched for values. This can be seen in $\Diamond$ algorithm 7 in the **if** statement beginning on line 2 and ending on line 7. This optimization is depicted in the dp_taliro algorithm for computing robustness. However, the TLTk $\Diamond$ operation contains two optimizations not shown in the dp_taliro algorithm as far as the author is aware. These optimizations help reduce calls to algorithms 4, 5, and 6. Since these are inside the loop that iterates through time steps, these multiply the time complexity of that loop. The loop iterates $|\tau|$ times; this causes a drastic increase in time complexity. The first optimization helps to lower calls to algorithm 6. It can be seen in algorithm 7 line 15. The **if** statement checks if the lower time bound is 0. If the lower time bound is 0 there is no need to search for where the lower bound index will be because it will always be the time step you are currently looking at. The second optimization can be seen in algorithm 7 line 19 through 28. This optimization lowers the amount of times algorithm 4 gets called to search all of the time steps between the time bound index. To do this, it uses the index of the previous max in the last time step calculated. The index of the max calculated in the previous time step is stored in max_index. max_index initializes as -1, so it is easy to tell if we are processing the first time step on line 19. If it is the first time step we have to call algorithm 4 to search the full robustness array between $lower_bound_index$ and $upper_bound_index$. However, on the next time step we can check and see if that time step moves out of frame on line 22. If the max_index is greater than the new $upper_bound$ it has moved out of frame and the full algorithm 4 needs to be called again with a frame being the data in the robustness array between $lower_bound_index$ and $upper_bound_index$. The frame moves backwards in time as the signal is processed. However, if that max has not moved out of frame we only have to perform algorithm 4 on the new data that is coming into frame, then check if the result from that call to algorithm 4 to the previous max. This can be seen in algorithms 8 and 7 lines 19 through 28, with line 19 being the check if it is the first iteration. Line 22 checks if the current min/max index has moved out of bounds. The default else statement on line 25 will run one of the

algorithms 4 or 5 on the incoming data if the max is still in frame.

Algorithm 4: find_max($robustness, start, end$)

Input: Robustness array $robustness$
Input: start number $start$
Input: end number end
Output: maximum index max_index

```
1  max ← robustness[0]
2  max_index ← 0
3  for i ← start to end do
4  |   if max < robustness[i] then
5  |   |   max ← robustness[i]
6  |   |   max_index ← i
7  |   return max_index
8  ]
```

Algorithm 5: find_min($robustness, start, end$)

Input: Robustness array $robustness$
Input: start number $start$
Input: end number end
Output: minimum index min_index

```
1  min ← robustness[0]
2  min_index ← 0
3  for i ← start to end do
4  |   if robustness[i] < min then
5  |   |   min ← robustness[i]
6  |   |   min_index ← i
7  |   end
8  |   return min_index
9  end
10 ]
```

Algorithm 6: search_sorted($\tau, start, end, time$)

Input: time array τ
Input: start index $start$
Input: end index end
Input: Target time $time$
Output: Middle index $middle$

```
1  lower_index ← start
2  upper_index ← end − 1
3  middle ← (lower_index + upper_index)/2
4  while lower_index ≤ upper_index do
5  |   if τ[middle] < time then
6  |   |   lower_index ← middle + 1
7  |   else if τ[middle] == time) then
8  |   |   break
9  |   else
10 |   |   upper_index ← middle − 1
11 |   end
12 |   middle = (lower_index + upper_index)/2
13 end
```

Algorithm 7: FINALLY(I,ε,τ,ς)

```
   Input: Time bound array of two numbers I
   Input: Robustness array ε
   Input: Time Step array τ
   Input: Result robustness array ς
   /* If time bounds are 0 and ∞ perform cheaper calculation */
 1 if I = [0, ∞] then
 2     max ← ε[|τ|]
 3     for i ← |τ| to 0 do
 4         if max < ε[i] then
 5             max ← ε[i]
 6         ς[i] ← max
 7 else
 8     max_i ← −1
 9     max ← −∞
10     for i ← |τ| to 0 do
11         time_l ← τ[i] + I[0]
12         time_u ← τ[i] + I[1]
13         index_u ← Search_sorted(τ, i, |τ|, time_u)
           /* Checks if search needs to be performed */
14         if I[0] == 0 then
15             index_l ← i
16         else
17             index_l ← Search_sorted(τ, i, |τ|, time_l)
           /* Does full search on first iteration */
18         if max_i == −1 then
19             max_i ← Find_max(ε, index_l, index_u)
20             ς[i] ← ε[max_i]
           /* check if the max has moved out of frame */
21         else if index_u < max_i then
22             max_i ← Find_max(ε, index_l, index_u)
23             ς[i] ← ε[max_i]
           /* runs if max has not moved out of frame.  Is cheaper than
              running full search */
24         else
25             possible_max_i ← Find_max(ε, index_l, prev_index_l)
26             if max ≤ ε[possible_max_i] then
27                 max_i ← possible_max_i
28         ς[i] ← ε[max_index]
29         prev_index_l ← index_l
30         max ← ε[max_i]
```

Algorithm 8: GLOBAL(I,ε,τ,ς)

Input: Time bound array of two numbers I
Input: Robustness array ε
Input: Time Step array τ
Input: Result robustness array ς

```
   /* If time bounds are 0 and ∞ perform cheaper calculation */
1  if I = [0, ∞] then
2      min ← ε[|τ|]
3      for i ← |τ| to 0 do
4          if ε[i] < min then
5              min ← ε[i]
6          ς[i] ← min
7  else
8      min_i ← −1
9      min ← −∞
10     for i ← |τ| to 0 do
11         time_l ← τ[i] + I[0]
12         time_u ← τ[i] + I[1]
13         index_u ← Search_sorted(τ, i, |τ|, time_u)
           /* Checks if search needs to be performed */
14         if I[0] == 0 then
15             index_l ← i
16         else
17             index_l ← Search_sorted(τ, i, |τ|, time_l)
           /* Does full search on first iteration */
18         if min_i == −1 then
19             min_i ← Find_min(ε, index_l, index_u)
20             ς[i] ← ε[max_i]
           /* check if the max has moved out of frame */
21         else if index_u < max_i then
22             min_i ← Find_min(ε, index_l, index_u)
23             ς[i] ← ε[max_i]
           /* runs if max has not moved out of frame. Is cheaper than
              running full search */
24         else
25             possible_min_i ← Find_min(ε, index_l, prev_index_l)
26             if min ≤ ε[possible_min_i] then
27                 min_i ← possible_min_i
28         ς[i] ← ε[min_index]
29         prev_index_l ← index_l
30         min ← ε[min_i]
```

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 HARDWARE AND SOFTWARE

All experiments were ran on the same computer that contained the following hardware and software:

- **OS:**

  Ubuntu 18.04

- **Processor:**

  Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz

- **GPU:**

  GeForce GTX 1080 Ti

- **RAM:**

  32 GiB of DDR4 RAM @ 2666MHz

These tests were ran in a non GUI environment to help remove external programs running.

5.2 SERIAL EXECUTION VS PARALLEL EXECUTION

In this sections we will be comparing results of the parallel execution and single core executions of TLTk. The results can be seen in table 5.1. Running robustness calculations of one dimensional formulas seems to have the greatest impact. With the greatest improvements see on φ_{b2} and φ_{b3} with the parallel version being five times faster at least. These tests uses 12 threads that the 6 core i7 processor can provide. The formula φ_{s1} shows a small speed up when threaded but not as drastic of one. This is probably because most of the time is spent on a slow memory copy from python to the quadratic programming solver.

Table 5.1: Threaded vs Unthreaded TLTk time in seconds

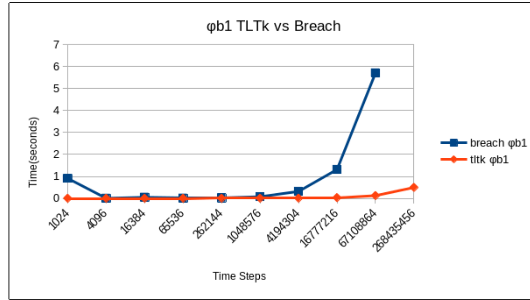
Size—threaded	φ_{b1}	φ_{b2}	φ_{b3}	φ_{s1}
1048576	0.0020	0.0146	0.0173	1.6824
2097152	0.0035	0.0294	0.0317	3.3175
4194304	0.0074	0.0587	0.0660	6.5242
8388608	0.0141	0.1289	0.1339	13.4716
16777216	0.0271	0.2544	0.2666	26.1716
Size—Unthreaded	φ_{b1}	φ_{b2}	φ_{b3}	φ_{s1}
1048576	0.0027	0.0920	0.0651	1.8909
2097152	0.0045	0.1934	0.1337	3.8003
4194304	0.0086	0.4078	0.2785	7.5991
8388608	0.0169	0.8598	0.5836	15.5394
16777216	0.0335	1.8129	1.2151	30.5196

Table 5.2: One dimensional formula list

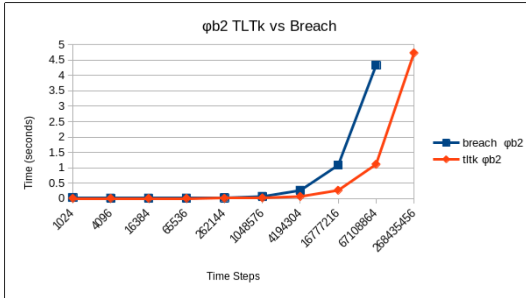
Specification	Predicates
$\varphi_{b1} = \neg(\Diamond s_1)$ $\varphi_{b2} = \neg(\Diamond_{[0,1000]} s_1 \wedge \Box_{[100,300]} r_1)$ $\varphi_{b3} = \neg(\Diamond_{[0,1000]} s_1 \wedge \Box_{[0,200]} (r_1 \wedge \Box(\Diamond(s_1 \wedge (s_1 U r_1)))$	$s_1 : speed(t) > 160$ $r_1 : rpm(t) < 4500$

5.3 TLTK VS BREACH

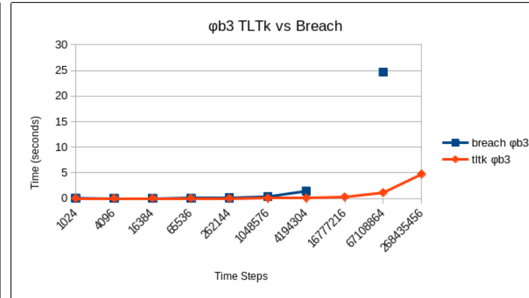
In this section we will look at TLTk vs breach. Breach is ran by calling matlab in terminal and running with the no display option. The results can be seen in table 5.2 Any cells with inf is where the program crashed. This was mostly do to a memory error due to running out of memory on the machine. In all experiments TLTk performed faster than breach.



(a) φ_{b1} execution time

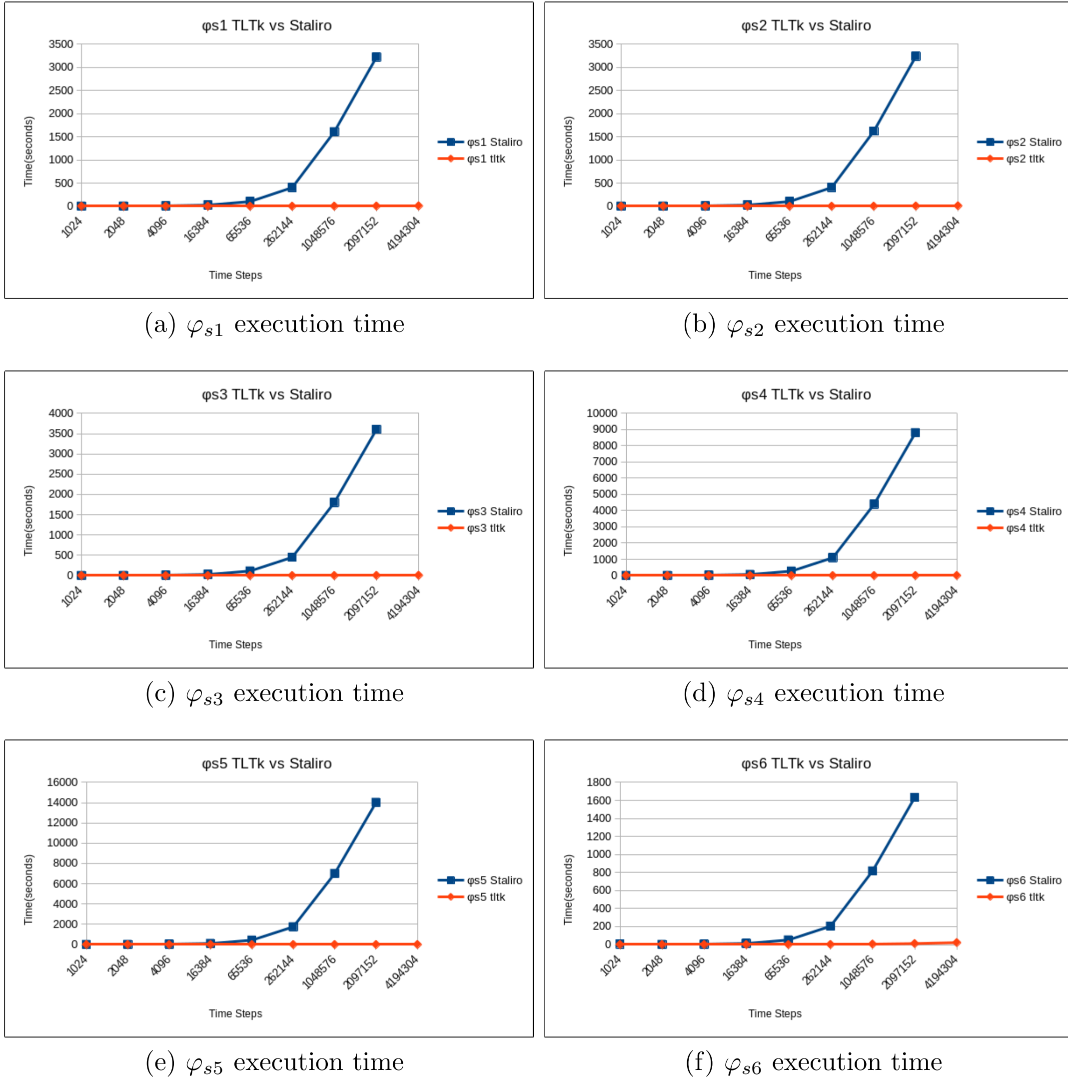


(b) φ_{b2} execution time



(c) φ_{b3} execution time

Figure 5.1: One dimensional execution time(seconds)



(a) φ_{s1} execution time

(b) φ_{s2} execution time

(c) φ_{s3} execution time

(d) φ_{s4} execution time

(e) φ_{s5} execution time

(f) φ_{s6} execution time

Figure 5.2: Higher dimensional execution time(seconds)

5.4 TLTK VS STALIRO

In this section we will be comparing Staliro to TLTk. We can see the large increase of Staliro's execution time in table 5.3. φ_5 is a larger formula than φ_3 which at 2^{21} samples have very different execution times. A cell marked with inf is where a crash accord. It is uncertian why staliro crashes at higher trace sized because the computer still had plenty of memory. In all experiments TLTk performed faster than Staliro.

Table 5.3: Larger Dimensional Formulas

MTL Specifications and Predicates
$\varphi_{s1} = \neg(\Box_{[5,150]} r_3 \wedge \Diamond_{[300,400]} r_4)$ where $r_3 : A_{s1} * x \leq \begin{pmatrix}250 & -240\end{pmatrix}^T$ and $r_4 : A_{s1} * x \leq \begin{pmatrix}240 & -230\end{pmatrix}^T$ and $A_{s1} = (1\ 0\ 0; -1\ 0\ 0)$
$\varphi_{s2} = (\neg p_{11}) U p_{12}$ where $p_{11} : A_{s2} * x \leq \begin{pmatrix}3.8 & -3.2 & 0.8 & -0.2\end{pmatrix}^T$ and $p_{12} : A_{s2} * x \leq \begin{pmatrix}3.8 & -3.2 & 1.8 & -1.2\end{pmatrix}^T$ and $A_{s2} = (1\ 0\ 0\ 0; -1\ 0\ 0\ 0; 0\ 1\ 0\ 0; 0\ -1\ 0\ 0)$
$\varphi_{s3} = \Box(r_7 \wedge \Diamond_{[0,100]} r_8)$ $\varphi_{s4} = \neg(\Diamond r_7 \wedge \Box(r_8 \wedge \Box(\Diamond(r_7 \wedge (r_7 U r_8)))))$ $\varphi_{s5} = \neg(\Diamond r_7 \wedge \Box(r_8 \wedge \Box(\Diamond(r_7 \wedge (r_7 U r_8)))) \wedge \Diamond(\Box(r_7 \vee (r_8 U r_7))))$ where $r_7 : A_{s345} * x \leq \begin{pmatrix}1.6 & -1.4 & 1.1 & -0.9\end{pmatrix}^T$ and $r_8 : A_{s345} * x \leq \begin{pmatrix}1.5 & -1.2 & 1 & -1\end{pmatrix}^T$ and $A_{s345} = (-1\ 1\ 0\ 0; 0\ 0\ -1\ 1)^T$
$\varphi_6 = \Box p$ where $p : eye(10) * x \leq (14.50\ 14.50\ 13.50\ 14.00\ 13.00\ 14.00\ 14.00\ 13.00\ 13.50\ 14.00)^T$

CHAPTER 6

CONCLUSION AND FUTURE WORK

The developed algorithm is able to take advantage of modern parallel processing to speed up execution time. The TLTk is able to process a large amount of data in a quick and scalable way. This will allow for MTL robustness to be used on data sets that need to have a large resolution of time steps or covers a large amount of time.

The main goal of the implementation of this algorithm was to find a balance between being efficient and easily expanded. With the the main interface being python, a new MTL operation could be added in a few lines of code. Once the new MTL operation is tested in python it can be made more efficient with language that compiles, such as C or C++.

This work provides the essentials for MTL robustness calculation. Future additions to this project are:

- **More MTL operations with C back ends:**

  The current implemented code has only six officially supported operations. There are many more MTL operations that people need.

- **Add a MTL string parser:**

  Currently, the only way to represent MTL formulas in tltk is calling MTL objects. There will be a parser that reads a string that represents a MTL formula and translates it in to the python objects.

- **differnt predicate types:**

  Currently, the only predicates that are supported are the distance between the polyhedron $Ax \leq b$ and a point $\mathbf{s}$. Future works will include the distance from shapes like ellipsoids. It will also allow for the signal to not be composed of points, but instead be composed of moving sets. This will allow the optimization problem to find the closest two points in two different sets.

REFERENCES

[1] H. Yang "Dynamic Programming algorithm for Computing Temporal Logic Robustness"

[2] G. E. Fainekos, H. Kress-Gazit and G. J. Pappas, "Hybrid Controllers for Path Planning: A Temporal Logic Approach," *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 4885-4890, 2005.

[3] Sarra Alqahtani, Ian Riley, Samuel Taylor, Rose Gamble, and Roger Mailler. 2018. MTL Robustness for Path Planning with A*. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '18). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 247–255.

[4] G. E. Fainekos and G. J. Pappa, "Robustness of temporal logic specifications for continuous-time signals *Theoretical Computer Science Vol 410, Issue 42*, pp. 4262 - 4291, 2009.

[5] Mishap Investigation Board, "Mars Climate Orbiter Phase I Report" November 10,1999

[6] R. Alur and T.Henzinger. Real time logics: complexity and expressiveness. *Fifth annual symposium on logic in computer science*, pages 390-401. IEEE Computer Society Press, 1990.

[7] A. Pnueli, "The temporal logic of programs," 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), Providence, RI, USA, 1977, pp. 46-57.

[8] GroBe, Daniel and Drechsler, Rolf. (2003). Formal verification of LTL formulas for SystemC designs. V-245 . 10.1109/ISCAS.2003.1206243.

[9] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, Model Checking, MIT Press, 1999, ISBN 0-262-03270-8

[10] S. Kirkpatrick, C. D Gelatt, Jr., M. P. Vecchi. Optimization by Simulated Annealing *Science Volume 220* pp, 4598

[11] A.Donze. Breach: A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. *In Computer-Aided Verification* pages 167-170, 2010

[12] TaLiRo Tools. [Online]. Available: https://sites.google.com/a/asu.edu/s-taliro/

[13] Y. S. R. Annapureddy. C. Liu, G. E. Fainekos and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Tools and algorithms for the construction and analysis of systems*, volume 6605 of LNCS, pages 254-257. Springer, 2011

[14] D. Goldfarb and A. Idnani(1983).A numerically stable dual method for solving strictly convex quadratic programs. Mathmatical Programming,27,1-33

[15] A. Donze and O Maler. Robust satisfaction of temporal logic over real-valued signals. In. Chatterjee and T. A. Henzinger, editors, FORMATS, volume 6246 of Lecture Notes in Computer Science, pages 92-106. Springer, 2010

[16] Fainekos, Georgios and Sankaranarayanan, Sriram and Ueda, K. and Yazarel, H.. (2012). Verification of automotive control applications using S-TaLiRo. Proceedings of the American Control Conference. 3567-3572. 10.1109/ACC.2012.6315384.

VITA

Graduate School
Southern Illinois University

Joseph Cralley

jkolecr@siu.edu

Southern Illinois University
Bachelor of Science, Computer Science, May 2018

Thesis Paper Title:
PARALLELIZED ROBUSTNESS COMPUTATION FOR CYBER PHYSICAL SYSTEMS VERIFICATION

Major Professor: Dr. H. Hexmoor