國立臺灣大學電機資訊學院電子工程學研究所

博士論文

Graduate Institute of Electronics Engineering
College of Electronical Engineering and Computer Science
National Taiwan University

Doctor Thesis

# 賽局上的合作時序邏輯的模型驗證以及密集錯誤回復力

# Model Checking for Temporal Cooperation Logic and Resilience against Dense Errors on Game Graph

黃重豪
Chung-Hao Huang

指導教授：王凡博士
Advisor: Farn Wang, Ph.D.

中華民國 105 年 1 月
January, 2016

國立臺灣大學
電子工程學研究
所

博士論文

# 賽局上的合作時序邏輯的模型驗證以及密集錯誤回復力

黃重豪　撰

105
1

# Abstract

This thesis is composed of 3 parts. In the first part, I introduce an algorithm to calculate the highest degree of fault tolerance a system can achieve with the control of a safety critical systems. Which can be reduced to solving a game between a malicious environment and a controller. During the game play, the environment tries to break the system through injecting failures while the controller tries to keep the system safe by making correct decisions. I found a new control objective which offers a better balance between complexity and precision for such systems: we seek systems that are k-resilient. A system is k-resilient means it is able to rapidly recover from a sequence of small number, up to k, of local faults infinitely many times if the blocks of up to k faults are separated by short recovery periods in which no fault occurs. k-resilience is a simple abstraction from the precise distribution of local faults, but I believe it is much more refined than the traditional objective to maximize the number of local faults. I will provide detail argument of why this is the right level of abstraction for safety critical systems when local faults are few and far between. I have proved, with respect to resilience, the computational complexity of constructing optimal control is low. And a demonstration of the feasibility through an implementation and experimental results will be in following chapters. The second part is to create an logic which can describe the different purposes of each player such as environment, controller, user, and etc in a system. I propose an extension to ATL (alternating-time logic), called BSIL(basic strategy-interaction logic), for the specification of strate-

gies interaction of players in a system. BSIL is able to describe one system strategy that can cooperate with several strategies of the environment for different requirements. Such properties are important in practice and I show that such properties are not expressible in ATL*, GL (game logic), and AMC (alternating μ-calculus). Specifically, BSIL is more expressive than ATL but incomparable with ATL*, GL, and AMC in expressiveness. I show that, for fulfilling a specification in BSIL, a memoryful strategy is necessary. I also show that the model-checking complexity of BSIL is PSPACE-complete and is of lower complexity than those of ATL*, GL, AMC, and the general strategy logics. Which may imply that BSIL can be useful in closing the gap between large scale real-world projects and the time consuming game-theoretical results. I then show the feasibility of our techniques by implementation and experiment with our PSPACE model-checking algorithm for BSIL. The final part is an extension to BSIL called temporal cooperation logic(TCL). TCL allows successive definition of strategies for agents and agencies. Like BSIL the expressiveness of TCL is still incomparable with ATL*, GL and AMC. However, it can describe deterministic Nash equilibria while BSIL cannot. I prove that the model checking complexity of TCL is EXPTIME-complete. TCL enjoys this relatively cheap complexity by disallowing a too close entanglement between cooperation and competition while allowing such entanglement leads to a non-elementary complexity. I have implemented a model-checker for TCL and shown the feasibility of model checking in the experiment on some benchmarks.

Key words:

# Contents

# Chapter 1

# Introduction

There can be million lines of code in today's software system. On such a scale of complexity, defects in the source codes are unavoidable. Various empirical studies show that the defect density of commercial software system is around 1 to 20 defects in every 1000 lines of source code [1]. Therefore, developers of systems created many engineering techniques to contain the damage that could be caused by such defects. For example, a software system may have several measures to its disposal to avoid system failure, including resending the request, resetting the server, clearing the communication buffers, and etc when observing that a critical service request is not acknowledged. However, in general, since all the recovery cost time and money it is important to estimate how to organize the measures for the maximal resilience of the system against realistic errors. At the moment, an automated support which can suggest defence techniques to development teams is missing. I created a game theoretic approach to study this problem and carried out experiments to show how this approach can be helpful in synthesizing the most resilient defence of software systems against multiple errors.

The naive way to measure the safety level of a system is to find the number of errors that it can endure before running into failure state. But in second thought, no non-trivial system can handle unlimited errors without degrading to inevitable system failure. Thus, it would be meaningless to analyse the resilience level of the systems to software errors can proceed without creating a realistic error model in which practical control mechanism can be devised to defend the systems against errors. In this work, I am interested in de-

fending the system against a more restricted error model, but still let the error model has a quantifiable level of power in order to simulate different error scenarios. Further more, I think a reasonable foundation need to take into consider that the life-time of a software system is much longer than the duration needed for a reasonably designed software system to recover from an error. Therefore, I propose to evaluate control mechanism of software systems on how many errors the control can endure before recovery to safe states. I then present an algorithm to find a control strategy that can handle the maximum number of such errors.

Let us standardize the basic terms before proceeding further. A design defect in software or hardware is called a *fault* in embedded systems. An *error* (sometimes called component failure in the literature) is the effect of a fault that causes a difference between the expected and the actual behavior of a software system, e.g., measurement errors, read/write errors, etc. An *error* does not always lead to a system failure, but may instead be repaired by, e.g., a defence mechanism in the software. That is, an *error* may be detected and fixed/neutralized before it creates any harm to the system or its users. A *failure* is the fact that users can observe the faulty behavior created by *errors*.

My specific goal is to develop a technique for finding a control mechanism of a software system which can against the maximal number of dense errors without degrading to failure. My inspiration is from methods for resilient avionic systems [2], where fault tolerance is designed to recover from a bounded number of errors. The number of errors a system needs to tolerate is calculated from the mean time between errors of individual components and the maximal duration of the system. I use the quality guarantees one obtains for an airplain(the system) as an example to demonstrate the difference between the objective to tolerate up to *k errors* and sequences of separated blocks of up to *k dense errors* in a short period. Assuming the operating time of the system is 20 hours, the mean time between exponentially distributed errors is 10 hours and the repair time is 3.6 seconds. The mean time between dense errors (consecutive errors before system recovery) is calculated in Table 1.1. The figures for $k$ errors (component failures) are simply the values for the Poisson distribution with coefficient 2. To explain the figures for $k$ dense errors,

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| $k$ errors | 0.865 | 0.594 | 0.333 | 0.143 | 0.053 | 0.017 | 0.005 | ... |
| $k$ dense errors | 0.865 | $2 \cdot 10^{-4}$ | $2 \cdot 10^{-9}$ | $2 \cdot 10^{-14}$ | $2 \cdot 10^{-19}$ | $2 \cdot 10^{-24}$ | $2 \cdot 10^{-29}$ | ... |

Table 1.1: Probabilities of $k$ dense errors

consider the density of 2 dense errors occurring in close succession. If an error occurs, the chance that the next error occurs within the repair time (3.6 seconds) is approximately $\frac{1}{10000}$. The goal to tolerate an arbitrary number of up to $k$-dense errors is, of course, much harder than the goal of tolerating up to $k$ errors, but, as the example shows, the number $k$ can be much smaller. Tolerating an arbitrary number of errors (with a distance of at least $3.6$ seconds between them) creates the same likelihood to result in a system failure as tolerating up to $9$ errors overall, and tolerating up to $15$ errors still results in a $70\%$ higher likelihood of a system failure than tolerating blocks of up to $2$ errors in this example. Only errors for which this is the case could cause a system failure. The mean time between blocks of two dense errors is therefore not ten hours, but 100,000 hours. Likewise, it increases to 1,000,000,000 (one billion) hours for blocks of three dense errors, and so forth.

Maximizing the number of dense errors that are permitted before full recovery is therefore a natural design goal. After full recovery, the system is allowed again the same number of errors. Now, if the *mean time between errors* (*MTBE*) is huge compared to the time the system needs to fully recover, then the mean time between system failures (MTBF) grows immensely.

We view the problem of designing a resilient control mechanism towards dense errors as a two-player game, called *safety resilience game*, between the system (protagonist[1], 'he' for convenience) and a hostile agent (antagonist[2], 'she' for convenience) that injects errors into the system under execution. The protagonist wants to keep the system from failure in the presence of errors, while the antagonist wants to derail the system to failure. Specifically, system designers may model their system, defense mechanism, and error model as a finite game graph. The nodes in the graph represent system states. These system states

[1] In game theory, a protagonist sometimes is also called *player 1*.
[2] In game theory, an antagonist sometimes is also called *player 2*.

are partitioned into three classes: the safe states, the failure states, and the recovery states. Some transitions are labeled with errors while others are considered normal transitions. The game is played with respect to a resilience level $k$. If a play ever enters a failure state, then the antagonist wins in the play. Otherwise, the protagonist wins.

The protagonists plays by selecting a move, intuitively the 'normal' event that should happen next (unless an error is injected). The antagonist can then decide to trigger an error transition (injecting an error) with the intention to eventually deflect the system into a failure state. Our error model, however, restricts the antagonist to inject at most $k$ errors before she allows for a long period of time that the system may use to recover to the safe states. (If the antagonist decides to use less than $k$ errors, the protagonist does not know about this. It proves that this information is not required, as we will show that the protagonist can play memoryless.) After full recovery by the protagonist to the safe states, the antagonist is allowed again to inject the same number of errors, and so forth.

If the system can win this game, then the system is called *$k$-resilient*. For $k$-resilient systems, there exists a control strategy—even one that does not use memory—to make the system resilient in the presence of blocks of up to $k$ dense errors. We argue that, if the component MTBF is huge compared to the time the system needs to fully recover, then the expected time for system breakdown grows immensely.

Besides formally defining safety resilience games, we also present algorithms for answering the following questions.

- Given an integer $k$, a set $F$ of failure states, and a set $S$ of safe states (disjoint from $F$), is there a recovery mechanism that can endure up to $k$ dense errors, effectively avoid entering $F$, and quickly direct the system back to $S$. Sometimes, the system designers may have designated parts of the state space for the recovery mechanism. The answer to this question thus also implicitly tells whether the recovery mechanism is fully functional in the recovery process.

- Given an integer $k$ and the set of failure states, what is the maximal set of safe states, for which the system has a strategy to maintain $k$-resilience? In game theory, this means that safety resilience games can be used for synthesizing safety regions for a

given bound on consecutive errors before the system is fully recovered.

The question can be extended to not only partition the states into safety, recovery, and failure states, but also for providing memoryless control on the safety and recovery states.

- Given a set of failure states, what is the maximal resilience level of the system that can be achieved with proper control? We argue that this maximal resilience level is a well-defined and plausible indicator of the defense strength of a control mechanism against a realistic error model.

With our technique, software engineers and system designers can focus on maximizing the number of dense errors that the system can tolerate infinitely often, providing that they are grouped into blocks that are separated by a short period of time, which is sufficient for recovery.

We investigate how to analyze the game with existing techniques. We present an extension to alternating-time $\mu$-calculus (AMC) and propose to use the AMC model-checking algorithm on concurrent games to check resilience levels of embedded systems. We present reduction from safety resilience games to AMC formulas and concurrent game structures. Then we present a PTIME algorithm for answering whether the system can be controlled to tolerate up to a given number of dense errors. The algorithm can then be used to find the maximal resilience level that can be achieved of the system. The evaluation is constructive: it provides a control strategy for the protagonist, which can be used to control a system to meet this predefined resilience level.

# Chapter 2

# Software Resilience against Dense Errors

## 2.1 Two-player concurrent game structures

## 2.2 Motivation

### 2.2.1 Background

### 2.2.2 Resilience in a Nutshell

## 2.3 Safety resilience games

## 2.4 Alternating-time $\mu$-calculus with events

### 2.4.1 Syntax

### 2.4.2 semantics

## 2.5 Resilience level checking algorithm

### 2.5.1 High-level description of the algorithm

### 2.5.2 Realization with AMCE mődel-checking

# Chapter 3

# Basic Strategy-interaction Logic

## 3.1 Existed Logics about Game and Strategy

### 3.1.1 Prior to Strategy Logics

### 3.1.2 With Strategy Logics

## 3.2 Running Example

### 3.2.1 Trying to Write Down a Correct Formal Specification

### 3.2.2 Resorting to General Strategy Logics for a Correct Specification

### 3.2.3 BSIL: New Strategy Modalities Expressive Enough for the Specification

### 3.2.4 Symbolic Strategy Names and Path Obligations

### 3.2.5 Passing Down the Path Obligations While Observing the Restrictions Among S-Profiles

### 3.2.6 Finding Finite Satisfying Evidence for a Formula in a Computation Tree

# Chapter 4

# Temporal Cooperation Logic

## 4.1 TCL

### 4.1.1 Syntax

### 4.1.2 semantics

## 4.2 Expressive Power of TCL

### 4.2.1 Comparison with CTL* and LTL

### 4.2.2 Comparison with AMC, ATL* and GL

## 4.3 Complexity

### 4.3.1 TCL Model-checking

### 4.3.2 TCL Satisfiability

## 4.4 Experiment

# Chapter 5

# Conclusions

# Bibliography

[1] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] John M. Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *FTRTFT*, volume 571 of *Lecture Notes in Computer Science*, pages 237–257. Springer, 1992.