Journal of Current Science and Technology, April - June 2025 Vol. 15 No. 2, Article 94
Copyright ©2018-2025, Rangsit University ISSN 2630-0656 (Online)

Cite this article: Pradubsuwun, D. (2025). Verification of Concurrent Go Programs using Timed Trace Theory. *Journal of Current Science and Technology*, *15*(2), Article 94. https://doi.org/10.59796/jcst.V15N2.2025.94

Journal of Current Science and Technology

Journal homepage: https://jcst.rsu.ac.th

Verification of Concurrent Go Programs using Timed Trace Theory

Denduang Pradubsuwun

Department of Computer Science, Faculty of Science and Technology, Thammasat University, Pathum Thani 12120, Thailand

E-mail: denduang@tu.ac.th

Received 6 August 2024; Revised 10 October 2024; Accepted 31 October 2024; Published online 25 March 2025

Abstract

The Go programming language, or Go, plays a critical role in developing concurrent programs because it provides features such as goroutines and channels that support program concurrency. Even though concurrency makes programs efficient, verification is required to ensure their correctness. This paper proposes a novel approach to verifying concurrent Go programs using timed trace theory. The proposed approach is specifically designed to verify concurrent systems. Verifying a Go program using timed trace theory is divided into two tasks: modeling and verification. Modeling involves transforming a Go program into time Petri nets using a proposed algorithm. Verification involves checking the conformance between the Go program and its specification. This can be done automatically by the timed trace theoretic verification tool, which supports a partial order reduction technique to mitigate the state explosion problem. We demonstrate the verification of the Philosopher problem using both the total order method and the partial order reduction method. Experiments with the Go program of the Philosopher problem demonstrate the effectiveness of the proposed method.

Keywords: *concurrent program; conformance checking; Go program; goroutine; time Petri net; timed trace theory; verify*

1. Introduction

Due to the advent of multicore processors, a program should be developed using a programming language that supports multithreading to take advantage of such processors. Go programming language or Go is one of the alternative languages for coding to meet this requirement. It provides features known as goroutines and channels for implementing concurrent programs (Meyerson, 2014). A goroutine is a lightweight thread and a channel is a pipe for communicating between goroutines, enabling them to send and receive values from each other. Although developing a Go program with concurrency makes the system efficient, i.e., (Fu, & Zhang, 2023; Gao et al., 2023; Hu, & Zhang, 2023), a verification of the Go program is needed for detecting failures to ensure that the Go program satisfies its specification. Verifying a concurrent Go program is challenging because its behavior is nondeterministic.

Typically, model checking is a popular method for verification. It is applied to verify several works, i.e., (Ghosh, & Karsai, 2023; Kitahara et al., 2022; Pang et al., 2024; Zhu, & Wang, 2023). It has been also proposed to verify Go programs, e.g., (Dilley, & Lange, 2021; Gabet, & Yoshida, 2020; Lange et al., 2018; Prasertsang, & Pradubsuwun, 2016). Model checking is a general-purpose verifier. However, a timed trace theoretic verification (Zhou et al., 2001), which is an extension of trace theory (Dill, 1988), was introduced. It was specifically designed to verify pure concurrent systems, e.g., asynchronous microprocessors. Applying timed trace theory to verify a system consists of two tasks. The first task is to convert the system and its specifications into time Petri nets, which is a formal model representing the concurrent system.

Secondly, the verification is done by an automatic tool based on timed trace theory, which is called a timed trace theoretic verification tool. Behind the tool, conformance checking is a technique for detecting failures. Likely, verifying the system may cause state explosion problems because the number of states is too large to handle. The timed trace theoretic verification tool also provides a partial order reduction (Pradubsuwun et al., 2005) to minimize the number of states for keeping away from the state explosion problems. On the other hand, the verification with all possible events is called the total order method.

In this paper, we propose a novel approach to verifying concurrent Go programs using timed trace theory. We also introduce an algorithm to transform a Go program into time Petri nets, which serve as input to the timed trace theoretic verification tool. We performed verification using both the total order method and the partial order reduction method.

2. Objectives

Our work proposes a novel approach to verifying concurrent Go programs using timed trace theory. To verify a concurrent Go program, two tasks must be performed: modeling and verification. Modeling involves transforming a Go program and its specification into time Petri nets. We propose an algorithm to perform this task. Verification involves checking the conformance between a Go program and its specification using timed trace theory. This verification can be performed automatically by the timed trace theoretic verification tool. The time Petri nets, which represent the Go program and its specification, serve as input to the timed trace theoretic verification tool. This tool also supports verification using partial order reduction. An overview of the proposed method is shown in Figure 1. We demonstrate the verification of a concurrent Go program for the Philosopher problem (Gabet, & Yoshida, 2020). The Philosopher problem involves a scenario in which multiple philosophers attempt to eat spaghetti simultaneously. The experiment is conducted by increasing the number of philosophers by one in each iteration, starting from two and ending with eight. Furthermore, both the total order method and the partial order reduction method are applied to verify the Go program for the Philosopher problem, allowing for a comparison of their efficiencies.

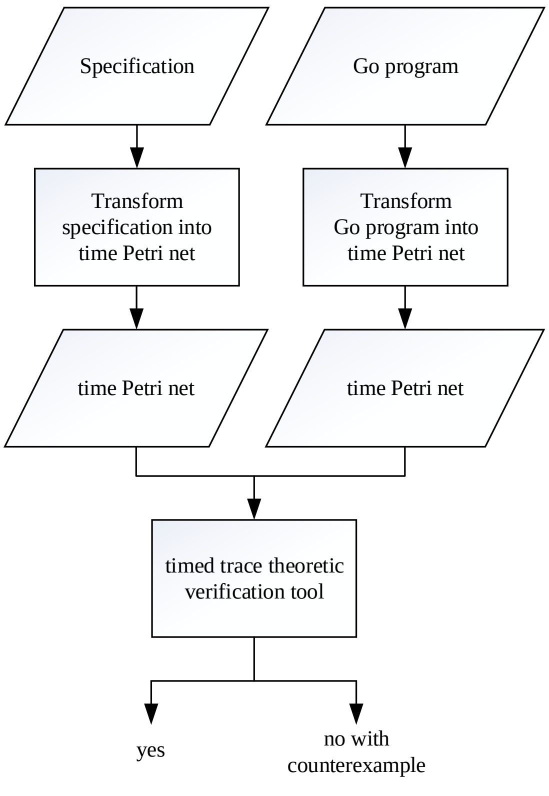

Specification
Go program
Transform specification into time Petri net
Transform Go program into time Petri net
time Petri net
time Petri net
timed trace theoretic verification tool
yes
no with counterexample

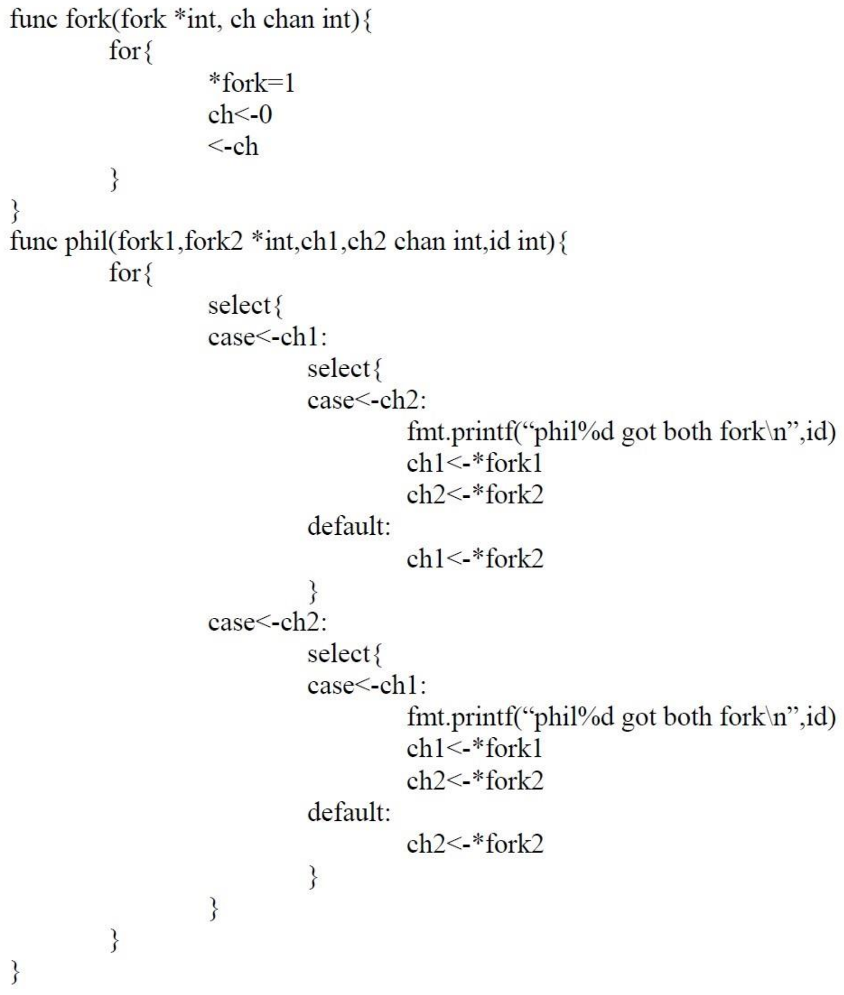
Figure 1 Overview of the proposed method

3. Materials and Methods

3.1 Modeling a Go Program

This subsection describes an algorithm to transform a Go program into time Petri nets. To understand the algorithm, we briefly describe a Go program and a time Petri net, respectively. A Go program (Go, 2019) may contain one or more functions. However, it must contain at least the main function, i.e., func main (). Each function is composed of a sequence of statements. There are several kinds of statements in Go. Since we are interested in the concurrent behavior of the Go program, we focus on control statements, e.g., if-else, for, select, and especially goroutines.

Goroutines are the important feature of Go. It is used to support the concurrent execution of functions in Go. For example, consider "go func()". The "go" keyword preceding a function call indicates that the function is executed concurrently. Moreover, Go provides a channel type, i.e., chan, for sending and receiving values between goroutines. This is accomplished using the channel operator. Before using the channel, it must be created using the make instruction. If there are multiple communication operations, the select statement is used to block until one of them can execute. Figure 2 shows an example of the Go program containing the philosopher function, i.e., "func phil()" and fork function, i.e., "func fork()" (Gabet, & Yoshida, 2020). The for {} statement is used in the body of both functions, representing an infinite loop.It represents an infinite loop. The func fork() function assigns a value of 1 to the variable fork and sends a value of 0 to the channel ch. The func phil() function sends the value of the fork1 parameter to the channel ch1 and the value of the fork2 parameter to the channel ch2 simultaneously.



```
func fork(fork *int, ch chan int){
        for{
                *fork=1
                ch<-0
                <-ch
        }
}
func phil(fork1,fork2 *int,ch1,ch2 chan int,id int){
        for{
                select{
                case<-ch1:
                        select{
                        case<-ch2:
                                fmt.printf("phil%d got both fork\n",id)
                                ch1<-*fork1
                                ch2<-*fork2
                        default:
                                ch1<-*fork2
                        }
                case<-ch2:
                        select{
                        case<-ch1:
                                fmt.printf("phil%d got both fork\n",id)
                                ch1<-*fork1
                                ch2<-*fork2
                        default:
                                ch2<-*fork2
                        }
                }
        }
}
```

Figure 2 An example of the Go program

Next, let us briefly explain the definition of a time Petri net. The behavior of concurrent systems can be represented by a time Petri net. A time Petri net consists of a six-tuple N = (P, T, F, lb, ub, u_0), where P is a set of places, T is a set of transitions, F is a set of flow relations specifying a binary relation between places and transitions ($F \subseteq (P \times T) \cup (T \times P)$), lb and ub are functions representing the earliest and latest firing times of transitions (i.e., lb: $T \rightarrow R^+$, ub: $T \rightarrow R^+ \cup \{\infty\}$), satisfying $lb(t) \leq ub(t)$ for all $t \in T$, and u_0 is an initial marking of the net. The time Petri net allows us to model both sequential and non-sequential behaviors of a system (i.e., conflicting and concurrent behaviors). The structure of a time Petri net differs depending on the behavior. This is illustrated in Figure 3: (a) sequential, (b) conflict, and (c) concurrent. Places are drawn as circles, transitions as bars, markings as solid circles, and flow relations as directed arcs. If every input place of a transition t contains a token, then t is enabled; otherwise, it is disabled. Each enabled transition t must fire within the time bounds lb(t) and ub(t). The firing of transitions represents the execution of the time Petri net, which is characterized by a state space defined by a set of inequalities.

Creating a time Petri net from a Go program is straightforward. We focus solely on the control flow of the Go program. Algorithm 1, shown in Figure 4, is proposed to transform a Go program into time Petri nets. The input of Algorithm 1 is a Go program and its output is a time Petri net implemented as a script file. Using the mapping file in Table 1, the control statements in the Go program are extracted as the flow relations of the time Petri net. The mapping file defines a correspondence between the control statements in Go (if-else, for, select, func, go) and the structure of the time Petri net. Similarly, when synthesizing the time Petri net for goroutines, we must take into account the channel, as the behavior of goroutines may be driven by it. Eventually, we obtain the time Petri net of the given Go program. Figure 5 shows the time Petri net of the GO program in Figure 2, which is derived by Algorithm1. Note that, according to the select statement in "func phil()", there are two cases. Both exhibit the same behavior, i.e., sending a value to channels ch1 and ch2 simultaneously. Thus, the time Petri net of "func phil()" is generated in only one case.

3.2 Timed Trace Theoretic Verification

Here, we explain the concept of timed trace theory (Zhou et al., 2001) for verifying systems. A system is represented as a module. The module M is defined as a tuple (I, O, N) where I is a set of input transitions, O is a set of output transitions, and N is a time Petri net. Its timed trace structure is denoted by T(M) = (I, O, S, F), where S is a success trace set, and F is a failure trace set. A trace y(w, t) is not in S (i.e., $y(w, t) \notin S$), for $w \in I \cup O$ and where t is the firing time of transition w, if and only if either (a) $y \in F$, or (b) $y \in S$, $t \leq TL(y, N)$, and $w \in I$, or (c) $y \in S$, $t > TL(y, N)$ and $limit(y, N) \subseteq I$, where TL(y, N) denotes the latest time until which the firing of all enabled transitions in N can be postponed after y, and limit(y, N) is the set of enabled transitions that determine TL(y, N). A specification is represented as a semimodule, M_s. The semimodule is the same as the module, except for the definition of its timed trace structure.

To verify whether a system satisfies its specification, a conformation between the module M and the semimodule M_s must be checked. Let us consider T(M) = (I, O, S, F) and $T(M_s) = (I_s, O_s, S_s, F_s)$, where $I_s = O$ and $O_s = I$, representing the timed trace structure of the module M and the semimodule M_s, respectively. The intersection of T(M) and $T(M_s)$ denoted by $T(M) \cap T(M_s)$, is a timed trace structure defined as $(I \cap I_s, O \cup O_s, S \cap S_s, (P \cap F_s) \cup (F \cap P_s))$ where $P = S \cup F$ and $P_s = S_s \cup F_s$. If $(P \cap F_s) \cup (F \cap P_s) = \emptyset$, then the module M conforms to the semimodule M_s. This implies that the system behaves in accordance with its specification in any failure-free environment. Here, the semimodule M_s serves as a maximum environment. In practice, conformance checking is performed by traversing the state space of $T(M) \cap T(M_s)$ to determine whether a failure exists. If the output produced by a (semi)module is not accepted by another (semi)module, a safety failure exists. If the input expected by the (semi)module is not provided on time by another (semi)module, a timing failure exists. This is a subset of liveness failures.

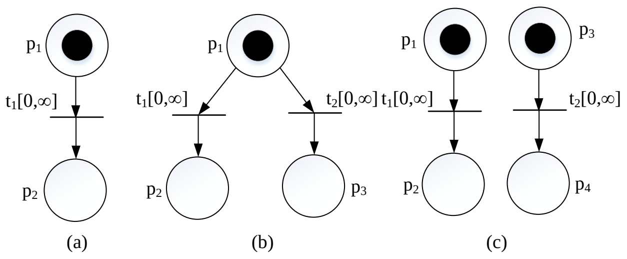


Figure 3 The structure of the time Petri net (a) sequential (b) conflict (c) concurrent

```
Algorithm1 Transform the Go program into the time Petri net.
Input:    the Go program
Output:   the time Petri nets $N=(P,T,F,lb,ub,u_0)$ or TPN file
1:        begin
2:          Initialize a TPN file as an empty file.
3:          Perform token analysis with the Go program and store a result in an
            intermediate code file.
4:          for each token in the intermediate code file do
5:                if token is in {func,go} then
6:                      M[i]=CreatetimePetrinet(token,M[i])
7:          Merge all M[i] to TPN file.
8:          return the TPN file
9:        end.
10:       CreatetimePetrinet(token,TPN file)
11:       begin
12:         for each token in the intermediate code file do
13:               if token is not in {func,go} then
14:               begin
15:                 Apply a mapping file to extract the flow relation F
                    corresponding to token.
16:                 Add places P, transitions T, and the flow relation F to the
                    TPN file.
17:               end
18:         Set the initial marking $u_0$ of TPN.
19:         return the TPN file
20:       end.
```

Figure 4 An algorithm to transform the Go program into time Petri nets

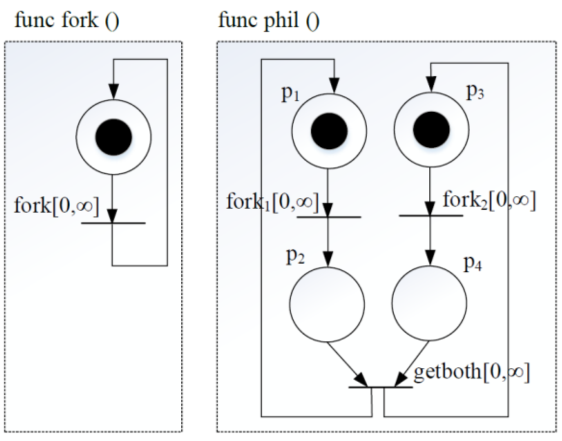
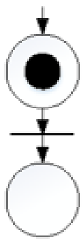
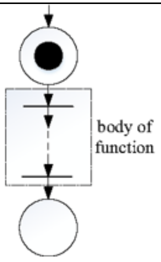
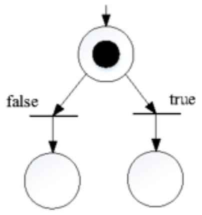
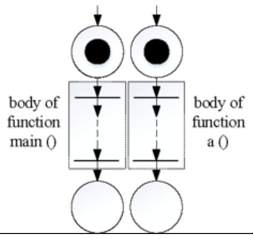
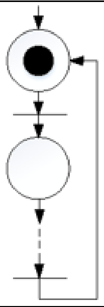
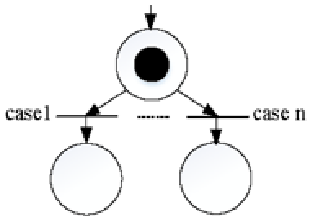


Figure 5 The time Petri net of the Go program in Figure 2

Table 1 A mapping from the Go statement to the time Petri net

Go statement	time Petri net	Go statement	time Petri net
sequential		Func	

If-else		Go func a () { }
		func main ()
		{
		go a ()
		}
			

For { }		Select	

3.3 Partial Order Reduction for Timed Trace Theoretic Verification

The timed trace theoretic verification tool provides an additional option for verification: a partial order reduction. It minimizes the number of states, thereby avoiding the state explosion problem. The idea of partial order reduction is to generate a subset of states in such a way that correctness remains unaffected. A state space constructed using partial order reduction is called a reduced state space, $G_r=(S_r,R_r)$, where S_r is a set of states and R_r is a set of transition relations between states. Note that (s,t_1,s_1) is a transition relation, meaning that s_1 is obtained from s by firing transition t_1. When the reduced state space G_r is constructed from the full state space $G_f=(S_f,R_f)$ where S_f is a set of states and R_f is a set of transition relations between states, it must satisfy the following three conditions (Pradubsuwun et al., 2005).

Condition 1: For $s \in S_r$, if s has successors in G_f, then s must have at least one successor in G_r.

Condition 2: For $s \in S_r$, if (s,t_1,s_1) and (s,t_2,s_2) are in conflict, and t_2 is not enabled in s then $(s,t_1,s_1) \in R_r$ implies that $(s,t_2,s_2) \in R_r$.

Condition 3: For $s \in S_r$, if (s,t_1,s_1) and (s,t_2,s_2) are concurrent, and the latest firing time of t_1 is greater than that of other transitions, then $(s,t_1,s_1) \in R_r$ implies that $(s,t_2,s_2) \in R_r$.

Condition 1 prevents the creation of a new deadlock state in G_r. Condition 2 handles conflict transitions. Since t_1 and t_2 are indirect conflict, the firing of t_2 may be missed if G_r contains only the firing of t_1. Therefore, the firing of t_2 must be included in G_r. Condition 3 addresses transitions that might conceal a timing failure. If, among concurrent transitions, t_1 has a later firing time than the others and relying solely on the firing of t_1 would mask a timing failure, then G_r must also include the firing of t_2.

3.4 Verifying the Go Program

Here, we demonstrate the verification of a Go program for the Philosopher problem, which is a concurrency-control problem. It involves philosophers eating spaghetti while sitting around a circular table. Each philosopher has his or her own plate. There is a fork between each plate. Philosophers can eat spaghetti only when both their left and right forks are available. These forks are not available when adjacent philosophers are eating. Thus, we must design a program that allows each philosopher to eat spaghetti continuously without a deadlock. Originally, five philosophers ate spaghetti concurrently. In this work, we verify the Go program for the Philosopher problem with a range of two to eight philosophers. We begin by verifying the Go program with two philosophers, then increase the number of philosophers by one in each iteration until there are eight philosophers.

The initial step is to model the Go program for the Philosopher problem as time Petri nets, as described in Section 3.1. Figure 6 shows the main function of the Go program with five philosophers (Gabet, & Yoshida, 2020), and Figure 7 shows the time Petri net of all "go phil()" statements from Figure 6 that are derived from Algorithm 1. The specification of the Go program shown in Figure 6 is illustrated in Figure 8. Note that the lower and upper bounds of time are considered to be 0 and ∞, respectively. In the next step, the time Petri nets of the Go program for the Philosopher problem and its specification are verified using the timed trace theoretic verification tool. Both the total order method and the partial order reduction method are applied to verify the Go program for the Philosopher problem. The verification results are presented and discussed in the next section.

```
func main(){
        var fork1,fork2,fork3,fork4,fork5 int
        ch1:=make(chan int)
        ch2:=make(chan int)
        ch3:=make(chan int)
        ch4:=make(chan int)
        ch5:=make(chan int)
        go phil(&fork1,&fork2,ch1,ch2,0)
        go phil(&fork2,&fork3,ch2,ch3,1)
        go phil(&fork3,&fork4,ch3,ch4,2)
        go phil(&fork4,&fork5,ch4,ch5,3)
        go phil(&fork5,&fork1,ch5,ch1,4)
        go fork(&fork1,ch1)
        go fork(&fork2,ch2)
        go fork(&fork3,ch3)
        go fork(&fork4,ch4)
        go fork(&fork5,ch5)
        time.Sleep(10*time.Second)
}
```

Figure 6 A main function of the Go program with five philosophers

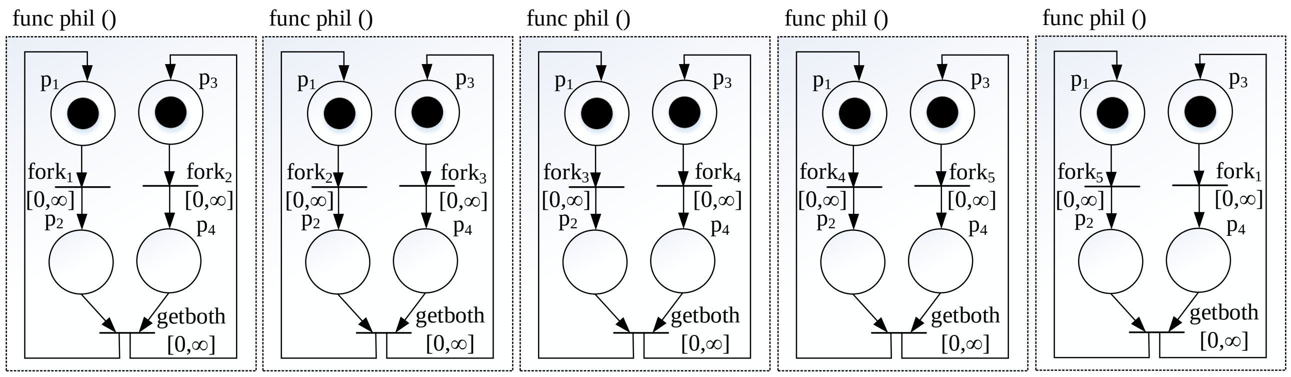


Figure 7 The time Petri net of all go phil() statements in Figure 6

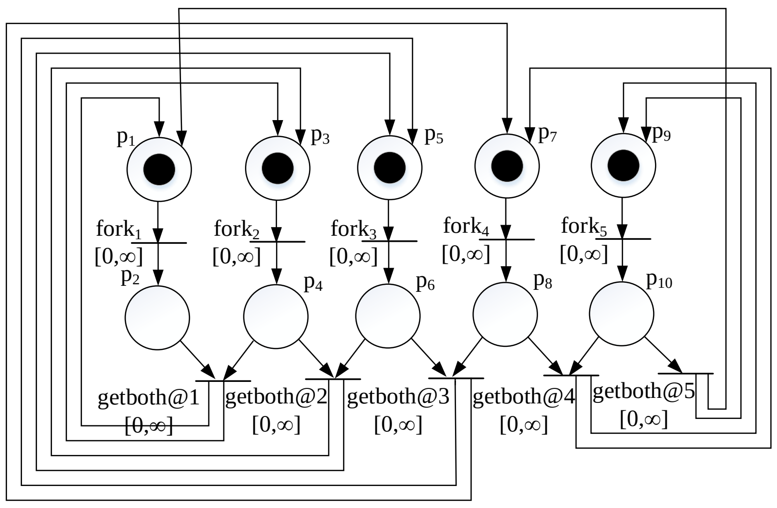
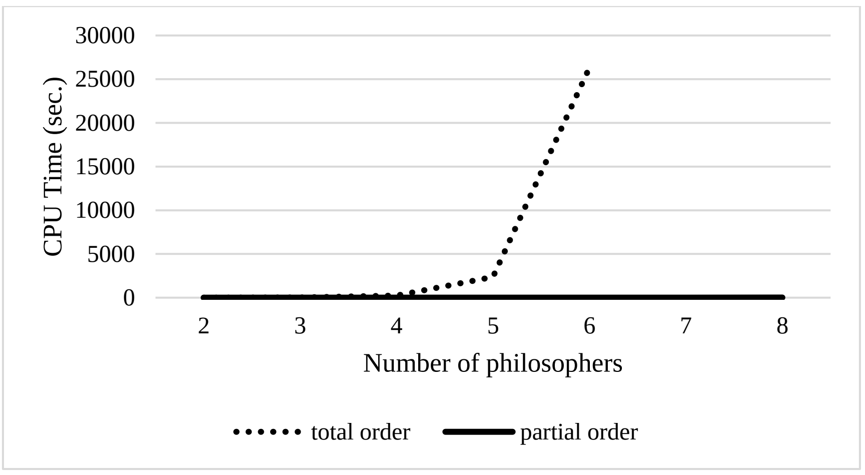
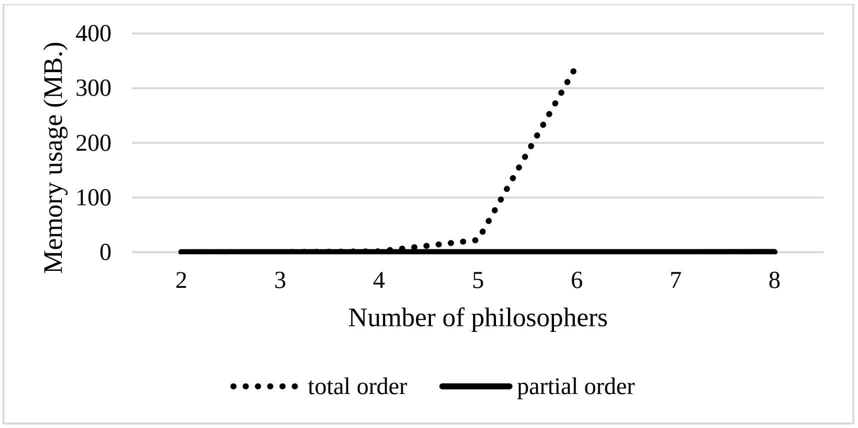


Figure 8 A specification of the Go program in Figure 6



(a)



(b)

Figure 9 (a) CPU times and (b) memory usage for verification of the Go program of the Philosopher problem

4. Results and Discussion

We verify the Go program for the Philosopher problem from two to eight philosophers. Initially, we transformed the Go program for the Philosopher problem into time Petri nets. Then, time Petri nets of the Go program and its specification are verified using the timed trace theoretic verification tool. We apply the total order method and the partial order reduction method to verify the Go program for the Philosopher problem. The experiments were conducted on a 2.6 GHz Intel Core i7 with 8 gigabytes of memory.

The verification results are illustrated in Figure 9. The CPU time and memory usage are measured because we focus on the resources used for verifying. The CPU time indicates how fast the verification is performed and the memory usage indicates the amount of memory required for verification. Figure 9(a) shows CPU times, and Figure 9(b) shows memory usage for verifying the Go program for n philosophers where $2 \leq n \leq 8$. In Figure 9(a), the x-axis represents the number of philosophers eating spaghetti simultaneously, and the y-axis represents the CPU time for verification. In Figure 9(b), the x-axis represents the number of philosophers eating spaghetti simultaneously, and the y-axis represents the memory usage for verifying. The solid line represents the verification using the total order method, the dashed line represents the verification using the partial order reduction method. The CPU time and memory usage increase dramatically when verifying the Go program using the total order method. Note that verifying with $n > 6$ by the total order method resulted in an out-of-memory condition. However, the CPU time and memory usage exhibit a scalable, non-explosive increase when verifying the Go program using the partial order reduction method.

The CPU time and memory usage for verification using the total order method increase significantly due to the increasing number of philosophers. This means that the greater the number of philosophers, the higher the level of concurrency. Verifying the Go program for the Philosopher problem using the total order method tends to cause a state explosion. On the other hand, the verification results show that verifying the Go program for the Philosopher problem using the partial order reduction method is far more effective. This is because the partial order reduction technique decreases the number of interleaving transitions in time Petri nets that must be checked by the timed trace theoretic verification tool. As a result, the number of states required for verifying is reduced. Verification using the partial order reduction method is an effective approach to handle such nondeterministic behavior in concurrent systems.

In comparison with the results of Gabet, & Yoshida (2020) for verifying the Go program for the Philosopher problem, their approach presents a model based on concurrent behavioral types and applies type-level model checking to detect failures. They do not address the state explosion problem, which is a critical issue in verifying concurrent systems. However, our work introduces a novel verification method for concurrent Go programs and demonstrates the efficiency of the partial order reduction method compared to the total order method. The timed trace theoretic verification using the partial order reduction method distinguishes our approach from theirs. It reduces the state space required for verification, thereby helping to avoid the state explosion problem. This represents a key advantage of our approach.

Moreover, the state explosion problem still persists due to the complexity of Go programs. When the number of philosophers increases, it adversely affects the overall performance of the verification method. Therefore, scalability emerges as a significant limitation of this work. Furthermore, since we demonstrate our verification method solely on the Philosopher problem, the generalizability of our findings is limited. Extending the method to more complex case studies is necessary. Hence, our approach can be applied to verify applications involving timed systems in concurrent environments, ensuring their correctness and compliance with time constraints.

5. Conclusions

In this work, we propose a novel approach to verifying Go programs using timed trace theory. The verification process is divided into two tasks. Firstly, we introduce an algorithm to transform a Go program into time Petri nets. A time Petri net is an appropriate formal model for representing goroutines that support the concurrent execution of the Go program. Secondly, the timed trace theoretic verification tool is applied to check the conformance between the Go program and its specification. This tool also supports the partial order reduction technique to minimize the number of states during verification. We demonstrated the verification of the Go program for the Philosopher problem using both the total order method and the partial order reduction method. We verified the Go program for the Philosopher problem with the number of philosophers eating spaghetti concurrently ranging from two to eight.

The experimental results demonstrate the superior efficiency of the partial order reduction method compared to the total order method. However, this work is limited in terms of scalability and generalizability. In the future, we plan to extend our verification method to more nontrivial case studies. These case studies may include more complex Go programs with intricate structures or programs with a variety of time bounds. This will help us address the limitations of this work.

6. References

Dill, D. L. (1988). Trace theory for automatic hierarchical verification of speed-independent circuits. *Advanced Research in VLSI: Proceedings of the 5th MIT Conference.* MIT Press. https://doi.org/10.7551/mitpress/1102.001.0001

Dilley, N., & Lange, J. (2021). *Automated verification of Go programs via bounded model checking* [Conference presentation]. 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), November 15-19, 2021, Melbourne, Australia. https://doi.org/10.1109/ASE51524.2021.9678571

Fu, S., & Zhang, K. (2023). *Network asset sensitive information management system based on Golang* [Conference presentation]. International Conference on Algorithms, Computing and Data Processing (ACDP), June 23-25, 2023, Qingdao, China. https://doi.org/10.1109/ACDP59959.2023.00030

Gabet, J., & Yoshida, N. (2020). *Static race detection and mutex safety and liveness for Go programs (extended version)* [Conference presentation]. European Conference on Object-Oriented Programming, Online Conference, November 15-17, 2020, Berlin, Germany. https://doi.org/10.48550/arXiv.2004.12859

Gao, C., Lv, H., & Tan, Y. (2023). *Multithreading technology based on Golang implementation* [Conference presentation]. 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), February 25-27, 2023, Shenyang, China. https://doi.org/10.1109/ACCTCS58815.2023.00116

Ghosh, P., & Karsai, G. (2023). *Distributed cyber physical systems software model checking using timed automata* [Conference presentation]. IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC), May 23-25, 2023, Nashville, TN, USA. https://doi.org/10.1109/ISORC58943.2023.00030

Go. (2019). *Documentation*. Retrieved July 2, 2024, from https://go.dev/doc

Hu, J., & Zhang, Y. (2023). *Design of remote monitoring system for ventilator based on Golang and MongoDB* [Conference presentation]. 6th International Conference on Artificial Intelligence and Big Data (ICAIBD), May 26-29, 2023, Chengdu, China. https://doi.org/10.1109/ICAIBD57115.2023.10206318

Kitahara, Y., Nakamura, M., & Sakakibara, K. (2022). *An investigation of formal verification of control policy of multi-car elevator systems using statistical Model checking* [Conference presentation]. International Conference on Machine Learning and Cybernetics (ICMLC), September 9-11, 2022, Japan. https://doi.org/10.1109/ICMLCp6445.2022.9941319

Lange, J., Ng, N., Toninho, B., & Yoshida, N. (2018). *A static verification framework for message passing in Go using behavioural types* [Conference presentation]. IEEE/ACM 40th International Conference on Software Engineering (ICSE), May 27-June 3, 2018, Gothenburg, Sweden. https://doi.org/10.1145/3180155.3180157

Meyerson, J. (2014). The Go Programming Language. *IEEE Software*, *31*(5), 101-104. https://doi.org/10.1109/MS.2014.127

Pang, S., Bian, Z., Zhang, Z., Meng, L., & Jiao, J. (2024). *A safety analysis method based on model checking* [Conference presentation]. 10th International Symposium on System Security, Safety, and Reliability (ISSSR), March 16-17, 2024, Xiamen, China. https://doi.org/10.1109/ISSSR61934.2024.00014

Pradubsuwun, D., Yoneda, T., & Myers, C. (2005). Partial order reduction for detecting safety and timing failures of timed circuits. *IEICE transactions on information and systems*, *88*(7), 1646-1661. https://doi.org/10.1093/ietisy/e88-d.7.1646

Prasertsang, A., & Pradubsuwun, D. (2016). *Formal verification of concurrency in Go* [Conference presentation]. 13th International Conference on Computer Science and Software Engineering (JCSSE), July 13-15, 2016, Khonkaen, Thailand. https://doi.org/10.1109/JCSSE.2016.7748882

Zhou, B., Yoneda, T., & Myers, C. (2001). *Framework of timed trace theoretic verification revisited* [Conference presentation]. Proceedings of the 10th Asian Test Symposium, November 19-20, 2001, Kyoto Japan. https://doi.org/10.1109/ATS.2001.990323

Zhu, W., & Wang, Y. (2023). *Model checking for scheduling on flight decks of aircraft carriers* [Conference presentation]. IEEE 6th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), February 24-26, 2023, Chongqing, China. https://doi.org/10.1109/ITNEC56291.2023.10082464