國立臺灣大學管理學院資訊管理學系

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

安全攸關軟體驗證方法與工具之研究

A Study of Methods and Tools for Verifying Safety-Critical Software

林靖婕

Jing-Jie Lin

指導教授：蔡益坤 博士

Advisor: Yih-Kuen Tsay, Ph.D.

中華民國 102 年 3 月

March, 2013

安全攸關軟體驗證方法與工具之研究

A Study of Methods and Tools for Verifying Safety-Critical Software

本論文係提交國立台灣大學
資訊管理學研究所作為完成碩士
學位所需條件之一部份

研究生：林靖婕　撰

中華民國一百零二年三月

謝辭

時光匆匆，台大資管的六年生活即將劃下句點，真正告別學生身分邁入社會的此刻，我想感謝這一路陪伴我走過的許許多多人。首先，感謝一直用心指導我們的蔡益坤教授，從大三的專題開始承蒙老師教導，在嚴謹的教學中，不僅是知識的傳授，更看到老師對學問的用心及熱忱，讓我受益良多。同時要感謝台大資管系的教授群，謝謝老師們的用心教學，六年間的累積，給予我邁步向前的力量與基礎。另外感謝王柏堯教授、陳郁方教授在口試中提供許多珍貴意見，讓此論文能更加完善。

此外，要感謝實驗室的大家。謝謝明憲學長總不斷指點迷津，不遺餘力幫助核研所計畫的進行，在學長身上真的學到很多。同樣謝謝啟祥在計畫的大力幫忙，也感謝兩年來一起修課的互相 cover。謝謝瑞舜和暐獻，在當演算法助教時麻煩你們很多，祝福你們論文都能十分順利。謝謝任峰學長、奕翔學長和辰旻學姐，無論是提供實驗室沙發或一起修課，感謝你們總是幫助我們許多，有你們在實驗室氣氛就格外歡樂。

另外也感謝資管系的所有同學，六年間共同擁有太多回憶，謝謝你們讓我的校園生活如此精彩，能認識你們真好。其中特別謝謝英瑛，謝謝妳總是在我心情低落時傾聽我的垃圾話，以後也請多多指教。最後要感謝我的家人，在求學路上一直給我最大的自由與支持，成為我永遠的後盾，謝謝你們讓我無後顧之憂的完成學業。

再次感謝伴我走來的大家，也期許自己能學以致用為社會貢獻，挺胸走出自己的道路。

林靖婕　謹識

于台大資訊管理學研究所

一百零二年三月

論文摘要

論文題目：安全攸關軟體驗證方法與工具之研究

作者：林靖婕　　102年3月

指導教授：蔡益坤　博士

安全攸關軟體的錯誤將導致人類傷亡，因此其正確性極為重要。安全攸關軟體典型為即時的多執行緒程式。多執行緒以執行多個並行活動，即時程式設計以確保滿足嚴格的時間限制。即時的多執行續程式設計易於犯錯，部分錯誤詭祕而難以用測試或模擬來發現。因此此類安全攸關軟體需要使用正規驗證方法加以檢查。

現今有許多方法及工具來支援即時多執行緒程式在功能及時間正確性上的正規驗證，每種方法或工具提供程式所涵蓋問題的部分解決方案。然而這些方法及工具由不同機構進行開發，因此選擇適合的工具或方法組合變得不易。為了實際驗證一程式，需要花費許多時間努力熟悉在不同領域的工具，並選擇適當的工具組合來完成驗證。在此論文中，我們對此類方法及工具做一精選，並介紹如何實行驗證。為使說明更易理解，我們以一具代表性的控制器程式為範例，此程式用以控制化學反應爐之溫度。論文中詳細說明功能正確性及時間正確性的驗證細節，並指出用以驗證的模型和實際程式間是否存在差異，以及是否存在現今工具無法支援的任務。我們以此論文建立一基準案例，透過此案例可迅速而容易地對現今即時多執行緒程式的正規驗證科技有一定的了解與評估。

關鍵詞：正規方法、模型檢查、多執行緒程式、即時系統、靜態分析、時間量測分析、最差情況執行時間

THESIS ABSTRACT

Graduate Institute of Information Management
National Taiwan University

Student: Lin, Jing-Jie Month/Year: March, 2013
Advisor: Tsay, Yih-Kuen

A Study of Methods and Tools for Verifying Safety-Critical Software

Safety-critical software is software whose failure harm people or even cause deaths, so the correctness of such software is very important. Safety-critical software is typically a real-time and multithreaded program. Multithreading is required because of multiple concurrent activities. Real-time programming is required to guarantee strict timing constraints. Real-time multithreaded programs are prone to mistakes, and some bugs in such programs are subtle and difficult to find using testing or simulation. Thus it is desirable to apply formal verification on such safety-critical programs.

Nowadays, there are many methods and tools that support formal verification of the functional and timing correctness of real-time multithreaded programs. Each method or tool provides parts of solutions to the issue involved. Unfortunately, the methods and tools have traditionally been developed separately by different communities, and it is nontrivial to assemble a suitable collection of them. To practically verify a program, one needs to spend much effort and time on getting familiar with the tools located in different domains, and to select an adequate tool collection to complete the verification tasks. In this thesis, we review a selection of methods and tools and show how they may be used to carry out the verification tasks. To provide a more comprehensive illustration, we consider a representative controller program as our target program, which is a temperature controller for chemical reactor protection. We delineate the details for verification of the functional correctness and timing correctness of the program. We also point out whether there exist differences between the model for verification and the real program, and whether there exist tasks that are not supported by current tools. In doing so, we establish a benchmark case and with it obtain a partial yet informative assessment of the readiness of current technology for formal verification of real-time multithreaded programs.

Keywords: Formal Methods, Model Checking, Multithreading, Real-Time Systems, Static Analysis, Timing Analysis, WCET.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Safety-critical software is the software that its failure will harm people or even cause deaths, so the correctness of the software is very important. Safety-critical systems arise in nuclear engineering, automotive, aviation, and spaceflight. And it is typically consists of real-time and multithreaded programs. Multithreading is required because of multiple concurrent activities Real-time programming is required to guarantee strict timing constraints Real-time multithreaded programs are prone to mistakes,and thus we rely more on methods and tools to ensure correctness of the code.

Code generation is one of the approaches, it generates source code through higher abstraction level codes that programmers specified. However, the tools of code generation has technical limitations that can not handle requirements at once, and it usually needs manual refinement, and thus it also need extra verification. Practically, the programs are developed from informal requirement descriptions to implementation code without comprehensive modeling or analysis. For those programs, we usually use verification to guarantee the correctness.

In practice, testing and simulation are still dominant, but research literature [54, 50] and industry standards [6, 3] have shown steadily increasing awareness of the advantages of using formal methods, including static analysis, model checking, and theorem proving. This applies to both design and code verification. For example, the current version of IEC 61508 [6] encourages the use of formal methods to help reduce test cases.

Real-time programs require to guarantee strict timing constraints, and real-time mul-

tithreaded programs have more complex timing behavior, thus it has to meet stringent functional and timing requirements. When coded as high-level programs with threads and timing functions, as now commonly done in the industry, their formal verification involves many different issues, some of which are rather complicated entailing difficult and tedious verification tasks. First of all, one should make sure that the code is free from any usual program safety error (so that it would not stop or crash unexpectedly). There is then the question of whether the program will carry out the intended functions correctly in the presence of concurrency. One may need to consider this both at the source and the object code levels. Probably even more tedious, if not harder, is whether all functions will be performed in a timely manner. This has to do with not only how the threads are arranged, but also which compiler is used and which operating system and processor the compiled code will be run on. All these together require support by a multitude of methods and tools.

1.2 Motivation and Objectives

We spent several weeks on relevant literature studying, but we could only find tools for some of the verification tasks. Each method or tool supports parts of solutions and proofs to issues and requires. Unfortunately, the methods and tools are traditionally been developed separately by different communities, and it is nontrivial to assemble a suitable collection of them. It is also challenge to identify the issues and find adequate tool meet the verification requires. To practically verify a program, we need to spend much effort and time on being familia with the tools locating in different domains, and need to select a adequate tool collection to completely verify. These may be attributed to the fact that the involved issues have traditionally been addressed and the methods and tools for their resolution developed separately by different communities.

To make more comprehensive illustration, we provide a trial case of a simple yet representative digital controller program, and delineate the various tasks entailed by its formal verification. The controller is part of a chemical reactor protection system and is implemented in the C programming language extended with pthreads and timing functions that are compatible with the real-time POSIX standard [7, 9, 8]. It represents

a typical piece of real-time software. Its main control-law computation involves tracking the temperature fluctuations of the reactor and is quite straightforward. The concurrent tasks as coded by the threads are periodic and their interactions are minimal.

There may be various verification tasks that can be done on the target controller. Instead of giving a comprehensive survey of all usable technologies, we review a "best-effort" selection of methods and tools and show how they may be used to perform the verification tasks. As we will show, certain tasks in the verification have to be carried out by hand because of the lack of suitable tools. There are also gaps between the abstract model and source code. In doing so, we establish the controller program as a benchmark case and with it obtain a partial yet informative assessment of the readiness of current technology for formal verification of real-time multithreaded programs.

1.3 Thesis Outline

The rest of this thesis is structured as follows:

- In Chapter 2, we give some preliminaries about this thesis, and introduce several related literatures.
- In Chapter 3, we introduce the controller and the requirements.
- In Chapter 4, we describe how to verify functional correctness of the controller.
- In Chapter 5, we describe how to verify timing correctness of the controller.
- In Chapter 6, we summarize our contributions, discuss the results of verification, also indicate some possible research direction in the future.

Chapter 2

Preliminaries and Related Work

In this section, we describe the structure of typical controllers, common features of real-time programming applied in such controllers, two main approaches in static program verification, timing and scheduling analyses of multi-task real-time systems, and cases of tools application.

2.1 Controllers and Real-Time Programming

A typical controller of a reaction plant basically has the following periodic tasks:

1. Receive sampled data from sensors. The data may be digitized by analog-to-digital converters (A/D).

2. Compute the control value from the sampled data, compare the desired value with the control value, and compute the output data. Trip the reaction and trigger alarms if the control value exceeds some preset limits.

3. Send the output data to actuators through digital-to-analog converters (D/A).

4. Communicate with a reliable network for sending data to and receiving data from remote controllers or remote sensors, or receiving control commands.

5. Check if there is any hardware failure.

6. Make sure by watchdog timers software modules running in this controller do not get stuck.

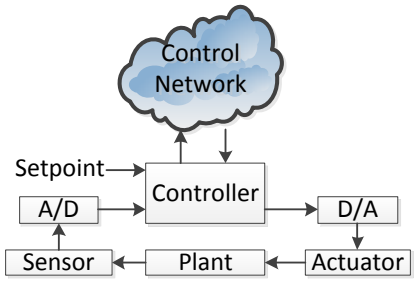

Figure 2.1: A typical control loop

It is rigorously required that the job of each periodic task initiated in one round must be finished before the next round.

The periodic tasks of a controller are usually implemented as threads synchronized with semaphores, mutex, or locks. To ensure the rigorous timing requirements can be satisfied, the controller usually runs on real-time operating systems (RTOS) which have less system operation overhead and more advanced scheduling algorithms. A periodic task can be programmed as a loop that processes the job first and then waits for the next round. The waiting time of each round may be different and can be calculated with the help of real-time clocks provided by the RTOS or by hardware equipments.

There are other common programming features applied in a controller. For example, low-level device access and network communication through sockets or message channels are required to interact with local I/O devices, remote hosts, and remote sensors; inter-process communication and synchronization may be used in a more complex controller; fault tolerance, redundancy, and failure detection are important to make the controller more robust.

2.2 Model Checking and Deductive Verification

The correctness of a program can be decomposed to several properties or specifications that should be satisfied by the program. There are two main approaches to the verification of functional correctness. One approach is model checking [21], which can be fully automatic but may not be able to prove undecidable properties. The other approach is deductive approach, which may need interaction with experts but can prove more properties than model checking.

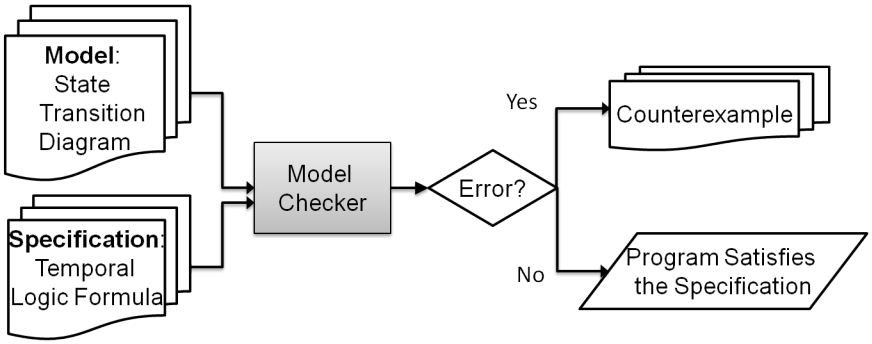

Figure 2.2: Basic idea of model checking

2.2.1 Model Checking

Figure 2.2 shows the structure of model checking. Facing a program, we can see the execution of the program as a transition of the states. So the program can be transformed to a state transition diagram, also a model. Specification is described with linear temporal logic (LTL) formula, which can assert how the behavior of the program evolves over time [21]. The model checker will check whether a model satisfy a LTL formula. If the model checker reports errors, the program violates the specification, and the counter example can show the error trail with interactive simulator. If there is no errors, the program satisfies the specification.

To check whether a model satisfy a LTL formula F, model checker generates an ω-automata A_F, which is equivalent to the LTL formula F. ω-automata is a finite state machine that run on words of infinite length, and it can be represented as a tuple $A = (Q, \Sigma, \delta, q0, Acc)$.

- Q is a finite set of states.
- Σ is the alphabet of A, is a finite set.
- $\delta : Q \times \Sigma \times Q$ is the transition relation.
- $q0$ is the initial set of states.
- Acc is the acceptance condition.

The set of accepted input strings is called the recognized ω-language by the automaton, denoted as $L(A)$.

After generating of A_F, the model checker generates the complement automata $A_{\sim F}$. The set $L(A_{\sim F})$ is the computing collection that can be identified by the complement, and the set $L(M)$ is the computing collection that can be identified by the model. If the intersection of $L(A_{\sim F})$ and $L(M)$ is not empty set, i.e. $L(M) \cap L(A_{\sim F}) \neq \emptyset$, the model violates the specification. Otherwise, the model satisfies the specification.

Verification of a program with model checking typically includes the following steps.

1. Construct an abstract model of the program.

2. Specify functional assertions.

3. Apply a model checker to search any assertion violation.

The construction of an abstract model from a program can be achieved either manually or automatically with predicate abstraction. The abstract model is usually represented as a state transition graph where a state carries the concrete or symbolic values of program variables at a specific time point in the program execution. The states of the abstract model may be finite or infinite. A program command corresponds to a state transition which changes the values of program variables. Note that system calls and third-party libraries used in the program should also be taken into account in the abstract model unless the adopted model checker can construct models from binary executables.

The functional assertions against the abstract model are usually expressed in boolean formulae, temporal logic formulae, automata, or other logic formulae. Any state that violates the assertions are identified as bad states. The goal of model checking is then to find bad states of the abstract model. In a bad state is found, the path from the initial state to the bad state is returned as a counterexample, which is very helpful in fixing the program. Standard safety properties are often checked by the model checker without additional assertions.

The state exploration by the model checker can be fully automated and is quite efficient since the state transition graph is already known. A main problem of model checking is state explosion, which means that the state size of the model is too large to be explored with limited memory size and CPU clockrate. During the past 10 years, several techniques were developed to tackle the state explosion problem. The techniques

include abstraction, partial-order reduction, symmetry reduction, composistional reasoning, symbolic model checking with binary decision diagrams and boolean satisfiability (SAT), counterexample-guided abstraction refinement (CEGAR) [20].

There are several efficient model checkers available, for example NuSMV [19], SPIN [36], CPAChecker [17], SLAM [16], ESBMC [23], Java Pathfinder [52], etc. While the model checker ESBMC supports threads, extensions of SPIN for multithread programs are also available [55].

And the tool Modex [11] is for automatic constructing the abstract model. Modex is short for Model Extractor. It can automatically transform a C program to a corresponding Promela model. For supporting automatic model extraction, SPIN versions 4.0 and later supports the inclusion of embedded C code by increasing five new primitives. The new primitives provide a powerful extension that let SPIN models have the full power of the arbitrary C code. SPIN can not check the embedded code fragments at either parsing or verification phase. The embedded code are directly copied from the model into verifier that SPIN generates. It is the C compiler that provides the required interpretation of all embedded code.

2.2.2 Deductive Verification

Deductive program verification is a method to prove the correctness of program through deductive reasoning. It accepts an annotated program and produces verification conditions that are required to prove the validity of the annotations, which specify as program specifications the pre-/post- conditions of functions or program segments, invariants of loops, and assertions at specified program locations. It is based on a program logic with axioms and inference rules, that capture the semantics of a program. Most deductive methods are based on Hoare logic [33] and predicate transformer [25].

Figure 2.3 shows the structure of deductive verification. First, user needs to annotate the program. The annotations are for expressing the pre-condition and post-condition of the fragment of the program and also the specification of verification. The program with annotations then will compile to an intermediate language. Doing so, it can support the input source code with several programming languages, and the verification condition generator thus can only designed for the intermediate language. For instance, the verifi-

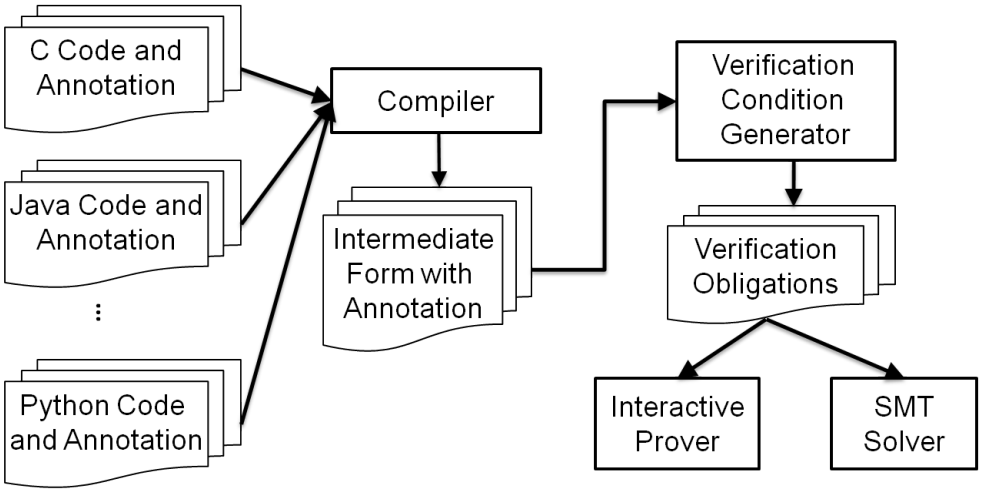

Figure 2.3: Workflow of deductive verification

cation tool Frama-C [4] compiles the program to C intermediate language(CIL). VCC of Microsoft also used the structure. [22]

The verification condition generator generates verification condition according to annotated intermediate language. If all of the verification properties can be proved, the origin program satisfies the specification that user asserted. The principle of verification condition generator bases on generating conditions that satisfy annotations from predicate transformer and weakest pre-condition. [25]

Below are typical verification steps of the deductive approach with a deductive verifier.

1. Rewrite the program features and syntax that are not supported by the verifier.
2. Model external functions such as system functions and third-party libraries by specifying their pre-conditions and post-conditions.
3. Annotate the program with specifications.
4. Run the verifier to generate verification conditions. Most verification conditions can be discharged automatically by decision procedures.
5. Prove unsolved verification conditions manually in interactive theorem provers.

For the verification of safety-critical systems, the behavior preservation of the rewritting program in step 1 and the model of system functions and libraries in step 2 should also be verified.

The program annotation and deduction are based on Hoare logic, which provides an axiomatic system to prove if a program satisfies its specification. The pre-/post-conditions of a program S is expressed by $\{P\}S\{Q\}$ where P is the precondition and Q is the postcondition. Both preconditions and postconditions can be expressed in some logic formulae, for example propositional logic, first-order logic, separation logic [49], etc. The annotation $\{P\}S\{Q\}$ is valid if and only if for every program state s that satisfies the precondition P, if the execution of S from state s ends at state s', then s' satisfies the postcondition Q.

The annotations of the program serve as the specifications that have to be satisfied by the program. Similarly, external functions such as system functions and third-party libraries used in the program also have their own specifications to be satisfied. Since external functions may not have source code available for establishing the validity of their specifications, the annotations of external functions are usually assumed. For safety-critical systems, the assumptions should be made as few as possible.

The verification conditions (or proof obligations) of the annotated program can be deduced based on Dijkstra's weakest precondition [25]. Depending on the logic adopted in the annotations, the generation of verification conditions is mostly automatic. Once a verification condition is generated, it may be proved and discharged immediately by automatic decision procedures such as Yices [26] and Z3 [47]. Verification conditions that cannot be discharged automatically are proved manually by experts with the help of interactive proof assistants such as Coq [51] and Isabelle/HOL [48].

A typical deductive verifier contains a verification condition generator and an automatic decision procedure. There are also several deductive verifiers, including Why [28], Frama-C [4], VeriFast [38], VCC [22], etc. The last two verifiers support threads while the first two verifiers do not.

2.3 WCET and Scheduling Analysis

The execution time of a task may vary, depending on the input and execution environment; the longest possible execution time is conventionally called the worst-case execution time (WCET) of the task. In Figure 2.4, the curve represents the distribution

Figure 2.4: Basic notions of WCET

of the number of the execution time that occurred when executing many times. Its minimum and maximum are the best-case execution time (BCET) and worst-case execution time (WCET).

To be assured that a task will be completed by the deadline, one must obtain a safe estimate of its WCET. Static WCET analysis is the primary method to obtain such safe estimates. A number of tools are available for static WCET analysis, including aiT [1], Bound-T [2], SWEET [13], etc.

Current static WCET analysis tools can only deal with sequential tasks. To obtain the worst-case response time (WCRT, which is the longest interval between the task's activation and the completion) of a task that contains concurrent activities or threads, the different threads need to be separated into individual sequential programs. Manual annotation of a program may be needed to help the tool to compute a more accurate WCET estimate. After obtaining the WCET's of all parts, a scheduling analysis can then be preformed to determine the WCRT of the task. Scheduling analysis can be divided into two phases [43] [32]: local scheduling analysis and compositional system level analysis. In the local scheduling analysis, the WCRT of a periodic task can be calculated by adding to its own WCET value the WCET's of all other tasks with higher priorities. In the compositional system level analysis, the WCRT of a task with cyclically dependent subtasks can be computed by using fix point methods. There are several tools for scheduling, including symTA/S [14], RT-Druid [12], etc.

There is another issue of a task containing both preemptive and non-preemptive jobs.

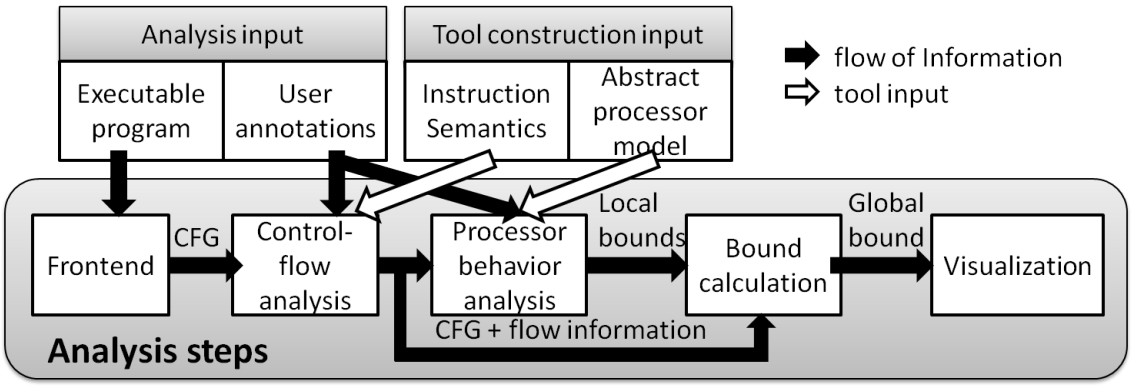

Figure 2.5: Components of a WCET tool that applied static method

When a job accesses a resource that is mutually exclusively shared with other jobs, the job enters the critical section and becomes non-preemptive. Preemptions cause jitters because of context-switch cost. One may use the UCB (Useful Cache Block) analysis [44] to calculate the jitters. If a memory block may be reached at program point P and if it may be reused at another program point reachable from P, it is called a UCB at program point P. Preemption may cause cache misses in memory blocks that are UCB's. The upper bound on the number of UCB's equals to the upper bound on the number of cache misses. The cache-related preemption delay can be calculated by multiplying the number of UCB's and the penalty.

2.3.1 WCET

Wilhelm et al. [53] proposed different approaches to calculate WCET and surveys several commercially available tools and research prototypes. There are two categories of approaches, static and measurement-based methods, and we introduce static one.

Static analysis does not execute the program on a hardware or simulator, but rather is based on abstract models of the hardware. Figure 2.5 shows the components and flow of information of a WCET tool that applied static method.

The analysis can be divided into several phases [53] [31]:

1. Value analysis, which determines the ranges of processor's registers and the values of local variables at each program point,

2. Control flow analysis, which computes the possible execution paths of the program,

3. Processor-behavior analysis, which predicts the cache, pipeline, and memory's influence on the execution time, and

4. Estimate calculation, which uses information obtained from the preceding phases to derive a WCET estimate.

In step of value analysis, it decides the bound of registers and the value of local variables at each program point. Usually, the results are not precise numbers, but are safety upper and lower bounds. Value analysis is important for cache analysis and loop bound analysis. It uses abstract interpretation: all the controller instructions are modeled as operations of abstract states, and the registers also correspond to possible interval by abstract states. [31]

Take `add` instruction as a example, D_{abs} is abstract register or memory cell, and $[l, u]$ represents the interval corresponding to a D_{abs}, l is the lower bound and u is the upper bound. The `add` instruction sums up the two D_{abs} to a new D_{abs}, and the interval of new D_{abs} is between the sum of lower bound of the two $D_{abs}(l_1 + l_2)$ and the sum of upper bound of the two $D_{abs}(u_1 + u_2)$. Due to the size of actual registers or memory cells are limited, we should check for the overflow problem. If overflow happens, the new interval of D_{abs} becomes unknown. [27]

$$add : D_{abs} \times D_{abs} \rightarrow D_{abs}$$

$$[l_1, u_1] + [l_2, u_2] = \begin{cases} [l_1 + l_2, u_1 + u_2], & \text{if no overflow is possible} \\ unknown, & \text{otherwise} \end{cases}$$

In step of control flow analysis, it collects all the possible execution paths. Due to the termination is granted, the set of path number is limited. We do not count the real sets of path, but instead count the superset of the real set. The superset is the safety approximation, and the smaller the better. Control flow analysis includes representation of input program, for example, call graph, control flow graph and other possible information as intervals of input data, bounds of execution times of loops. Information is from value analysis or user. The output of result can be viewed as the limitation of dynamic behavior of the program, including which functions will be called, relationships between conditions, and feasibility of the paths. [31]

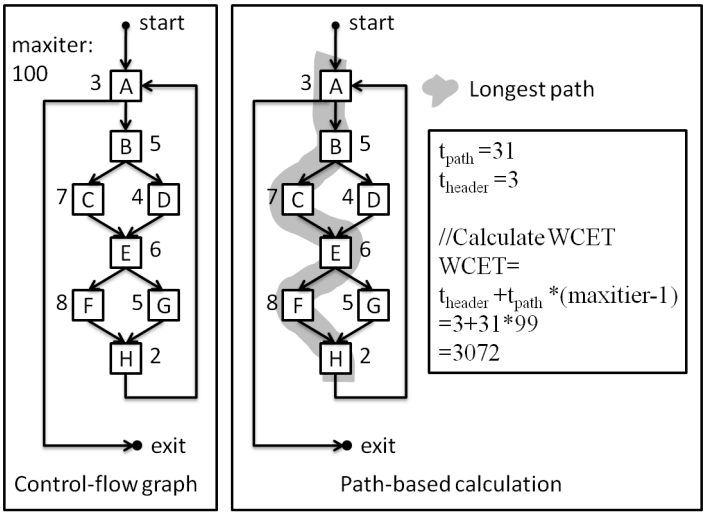

Figure 2.6: Path-based approach of estimate calculation [31]

In step of processor-behavior analysis, the processor includes many elements that make execution time contex-dependent, for example, memort, cache, pipeline and branch forecast. The execution time of a single instruction is dependent with its execution history. In order to find the precise bound of execution time, we need to analyze all the states of the elements of the processor that reach the path of the instruction. In this step, we decide the invariants of occupancy states. The tool should consider all the surrounding conditions of the processor, including the full memory hierarchy, the bus, and peripheral units. In general, the upper bound of execution time of a instruction is decided by the states of processor at the moment that executing the instruction. And the states of processor is decided by potential execution histories. Some of the instructions may be executed at specific condition, so the execution time may differ, the accuracy of calculation may be influenced. We can consider separate execution histories according to the context of information flow. [31]

In step of estimate calculation, it decides the estimation of the WCET. There are three approaches: path-based, structure-based, and implicit path enumeration technique (IPET). Figure 2.6 shows the path-base method. It calculates the upper bound of execution time of different paths, and find the longest execution time to be the upper bound of the task. [31] Figure 2.7 shows the IPET method. It combines data flow and upper bounds of execution time of basic blocks to sets of arithmetic constraints. Each basic

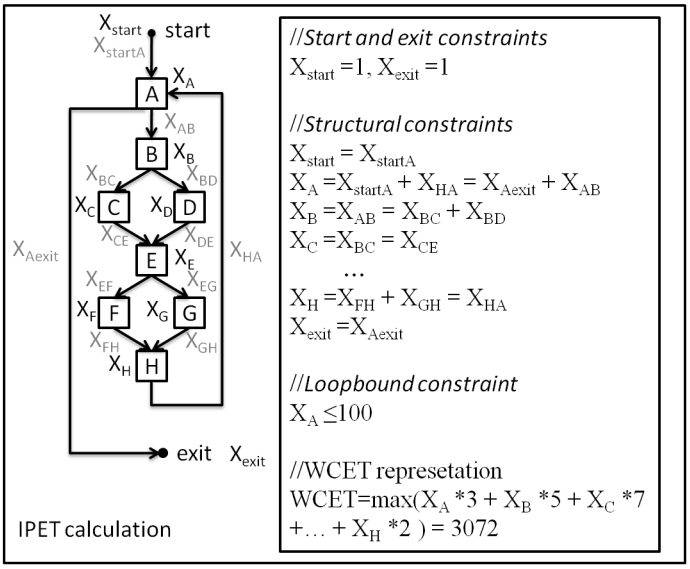

Figure 2.7: IPET approach of estimate calculation [31]

block and program flow edge has a time coefficient t_entity, representing the upper bound of contribution of the total execution time. The variable x_entity represents the number of execution time of the entity. The upper bound is determined by sum of the products of the number of execution time and the upper bound of contribution of the total execution time.($\sum_{i \in entity} x_i \times t_i$)

Figure 2.8 shows the structure-based method. It calculates the upper bound by bottom-up traversal of the syntax tree of the task, and the bounds of statements is computed by combination rules for that type of statement.

2.3.2 Scheduling Analysis

When several tasks share the same resources, scheduling is needed to prevent the conflicts of the requirements of resources,. Scheduling decides the order of the execution of the tasks, and the order that tasks getting the resources.

Local Scheduling Analysis

Tasks is activated by events. For instance, expiration of a timer, internal or external interrupt, and task chaining. Scheduling analysis abstracts individual activating events to event stream, and according to event stream to generate the scenarios of worst condition. Scheduling analysis calculate the worst-case response time (WCRT, the interval between

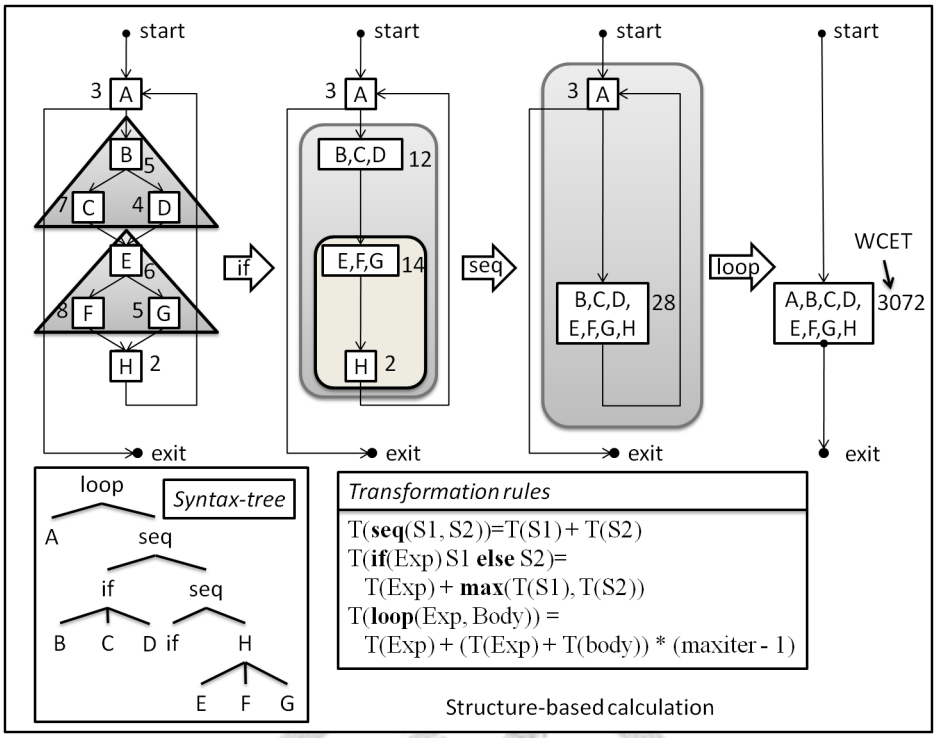

Figure 2.8: Structure-based approach of estimate calculation [31]

activation and termination) of the tasks through the scenarios.

Henia et al [32] proposed the model applied in SymTA/S [14]. Figure 2.9 shows the model of two tasks that share two resources. R1, R2 represents the tasks, and Src1, Src2 represents the resources, and E1 E4 represents the events. The model is called event model, which capture possible timing of activating events.

Event model is described with parameters. For example, (P, J) represents a periodic with jitter event model. P is the period and J is the the interval that may deviate from

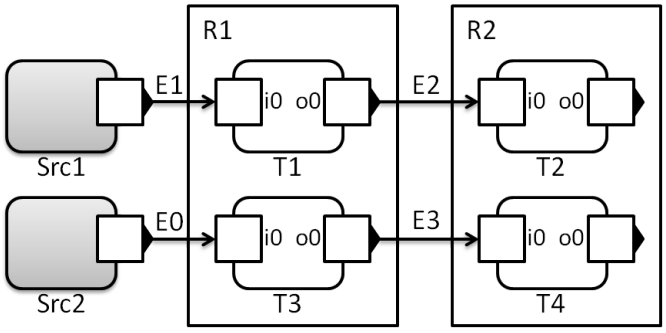

Figure 2.9: The model of two tasks that share two resources [32]

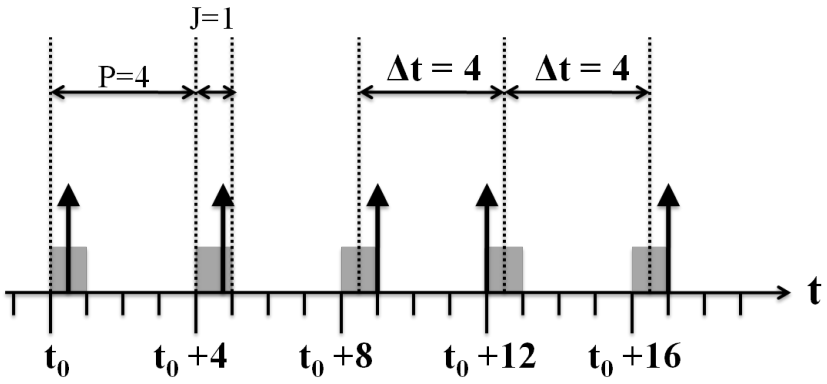

Figure 2.10: A periodic with jitter event model [32]

its origin location because of jitter. Figure 2.10 shows a example that $(P, J) = (4, 1)$, and every activation events will fall inside the gray boxes.

Upper event function $\eta^u_{P+J}(\Delta t)$ represents the upper bound of the number of events within interval Δt. Lower event function $\eta^l_{P+J}(\Delta t)$ represents the lower bound of the number of events within interval Δt.

$\eta^u_{P+J}(\Delta t) = \lceil \frac{\Delta t+J}{P} \rceil$

$\eta^l_{P+J}(\Delta t) = max(0, \lfloor \frac{\Delta t-J}{P} \rfloor)$

Event functions are piecewise constant step functions that with unit-height steps. Figure 2.11 shows the event function for $(P, J) = (4, 1)$. The black points indicate that upper event function use the point of small value, and lower event function use the point of large value. When in the interval of Δt, the number of events will limit in the scope between the upper and lower event function.

The minimum distance function $\delta^{min}(N \geq 2)$ represent the minimum distance between $(N \geq 2)$ consecutive events in an event stream, and maximum distance function $\delta^{max}(N \geq 2)$ represent the maximum distance.

$\delta^{min}(N \geq 2) = max\{0, (N-1) * P - J\}$

$\delta^{max}(N \geq 2) = (N-1) * P + J$

If the event is sporadic, the lower event function $\eta^l_{P+J}(\Delta t)$ is always zero, and $\delta^{max}(N \geq 2)$ is infinity for all values of N. With all the functions above, we can calculate the worst-case response time of a task by the following equation. [39]

$r_i = C_i + \sum_{\forall j \in hp(i)} (\eta^u_j(r_i)) * C_j$

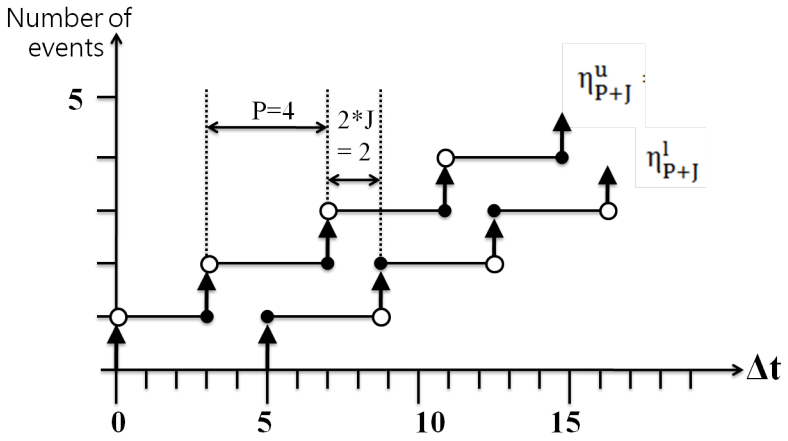

Figure 2.11: Upper and lower event functions [32]

r_i represents the WCRT of task i, and j represents the tasks that priorities are higher than task i. $hp(i)$ is a set that includes all of the tasks that priorities are higher than task i. C_i and C_j is the WCET value of task i and j. $\eta^u_j(r_i)$ represents the upper bound of the number of events of task j within interval r_i. The equation calculates the response time of task i under the largest number of interruptions by other higher priority tasks. Therefore, it is worst-case response time of task i.

Compositional System Level Analysis

In local scheduling analysis, when each task acquired the WCRT of itself, it will output a event model, as the activating event model of next task. The period of the output event model is the same as the activating event model, and the jitter of the output event model need add the interval between the shortest and longest response time.

$J_{out} = J_{act} + (t_{(resp,max)} - t_{(resp,min)})$

J_{out} is the output event model, and J_{act} is the activating event model. $t_{(resp,max)}$ is the WCRT, and $t_{(resp,min)}$ is the best-case response time.

Take Figure 2.9 as an example, the activating event models of $R1$ are all available, so it can get the WCRT($T1$, $T3$) and output event models through local scheduling analysis. The output event models are the activating event models of $R2$, and again it can get the WCRT($T2$, $T4$) through local scheduling analysis.

However, not all the situations can work as above, for instance in Figure 2.12, only

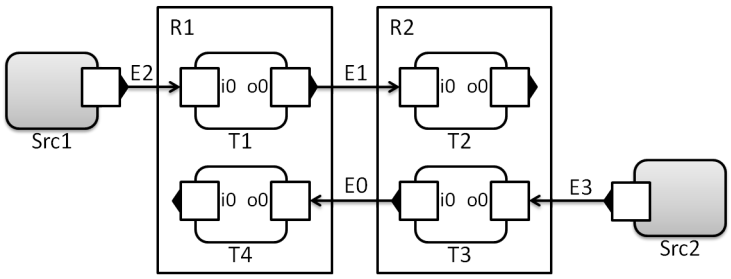

Figure 2.12: The model that two tasks shared two resources and with cycle [32]

the activating modelS of $T1$ and $T2$ are available, so the analysis can not work because not all the resources are available of a task. It need the WCRT of R2 to compute the WCRT of R1, and vise versa. The problem is called as cyclic scheduling dependency.

To solve the problem, it sends all the output event models to all the system path in the initial until the activation event models for every tasks are available. The method is safe because the scheduling do not change the period of event model, it only increase the jitter, and the larger jitter interval includes the smaller jitter interval, which is assumed to be safe.

After sending the output event model, it do compositional system level analysis. First, do local scheduling analysis for each task, when all the local scheduling analyses are finished and all the output event models are sent, check if there is activation event models need updated. If so, the output event models for that task may be changed, so do the local scheduling analysis again. If all the activation event models need not to be updated, it reaches the convergence. The last computed WCRTs are valid. If the stop conditions are satisfied in the analysis, e.g violate the scheduling limitation, the system Level analysis will stop. Figure 2.13 shows the process of compositional system level analysis.

2.4 Related Work

In this section, we review some cases of tools application. We classify the cases to three categories: applied tools for functional correctness, for timing correctness, and for the both.

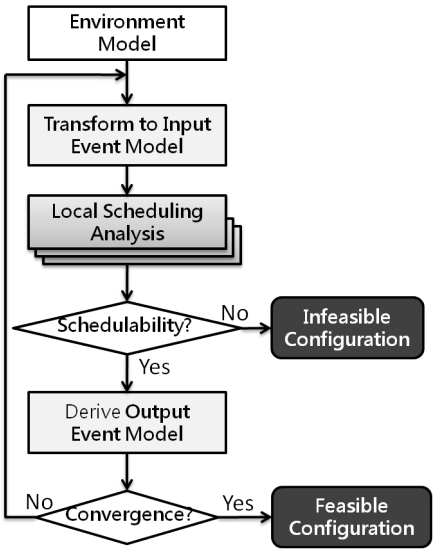

Figure 2.13: The process of compositional system level analysis [43]

2.4.1 Cases of Tools Application for Functional Correctness

Glück and Holzmann [29] applied SPIN+Modex [11] to check for flight software verification. The flight software is from NASA's Deep Space One (DS1) mission, and it is implemented in C. They used the model checker to find some defects at launch.

Kosina [42] used many tools to verify the code of Linux Kernel. For example, `Coverity`, `Sparse` and `MOPS`. He also used SPIN+Modex to find vulnerability of missing locking in function *load_elf_library*(). But the details is not shown in the paper.

Kim et al. [41] detected of concurrency bugs in the kernel by combining both model checking techniques with SPIN+Modex and testing methods. It proposed the MOdel-based KERnel Testing (MOKERT) framework, which includes Modex and SPIN. The advantage of the framework is it can replaying a counter example on the actual kernel code. The framework can prevent the counter example that was detected from the model checker is a false alarm, which is caused from the gap between abstract model and source code.

Hossain and Chowdhury [37] provided a practical approach on Model checking with Modex and SPIN. The target is a simple mutual exclusion program without using *mutex*. It showed and explained the details of process in the verification.

Cuypers, Jacobs, and Piessens [24] showed how VeriFast can be used to verify the

data-race-freedom of a multithreaded Java application.

2.4.2 Cases of Tools Application for Timing Correctness

Gustafsson and Ermedahl [30] summarized five different industrial case-studies that using WCET analysis. The first is for disable interrupt (DI) regions in a RTOS with the tool SWEET. The second is for DI regions and system calls in a RTOS with the tool aiT. The third is for automotive communication code with the tool aiT. The forth is for welding systems code with the tool aiT and a measurement tool oscilloscope. The fifth is for articulated haulers code with the tool aiT and an in-circuit emulator for measurement Finally, it discussed advantages and disadvantages of different timing analysis methods used.

Byhlin et al. [18] applied WCET analysis on automotive communication software. It verified the time-critical code in products from Volcano Communications Technologies AB (VCT). They investigate the practical difficulties that arise when applying current WCET analysis methods to particular kind of systems. aiT[1] is the WCET analysis tool they chose to verify.

Kästner et al. [40] presented a tool flow for validating timing behavior based on aiT [1] and SymTA/S [14]. Using XML Timing Cookies (XTC) to communicate SymTA/S s with aiT. The target they verified is embedded hard real-time systems. Moreover, they discussed the most important hardware components affecting timing predictability and summarizes their effect on the applicability of measurement-based approaches and on the efficiency and precision of static analysis methods.

2.4.3 Cases of Tools Application for Both Functional and Timing Correctness

Souyris et al. [50] verified avionics software products. Avionics software products are developed according to very stringent rules imposed by the DO-178B standard [3]. They used two kinds of formal techniques, deductive methods and abstract interpretation based static analysis for the verification. The tools they considered for deductive methods are Caveat and Frama-C [4]. The abstract interpretation based tools are Astree, aiT[1], Stackanalyzer and Fluctuat.

Chapter 3

Challenge Case: The Controller

3.1 Introduction of the Controller

Our verification target is a digital controller that controls the temperature of a chemical reactor to avoid thermal runaway by adjusting the flow rate of coolants passing through the reaction process. The temperature of the reaction is measured by the average of the temperature obtained from two sensors. If the temperature rises consecutively three times in the most recent measurements and the current temperature exceeds a specified high level, the controller increases (with output 1) the flow rate of coolants. Similarly, the controller decreases (with output -1) the flow rate of coolants if the temperature goes down consecutively three times in the most recent measurements and the current temperature is below a specified low level. It is safety-critical that the temperature cannot exceed a specified upper limit, which is higher than the high level. If the safety criteria is violated, the controller triggers a trip to stop the chemical reaction.

The controller is designed to run on systems compatible with real-time POSIX and is implemented in C. The industrial coding standard MISRA-C [15] is followed as much as possible. The basic controller functions are implemented by the five tasks shown in Figure 3.1: temperature control loop (TCL) task, sender task, receiver task, diagnosis task, and watchdog task. Each task is executed periodically in a POSIX thread. To make sure that the tasks do not get stuck, the controller comes with a hardware watchdog timer that should be reset periodically. If the hardware watchdog timer times out, the controller will be restarted.

The main thread initializes shared variables, hardware handles, and socket connections

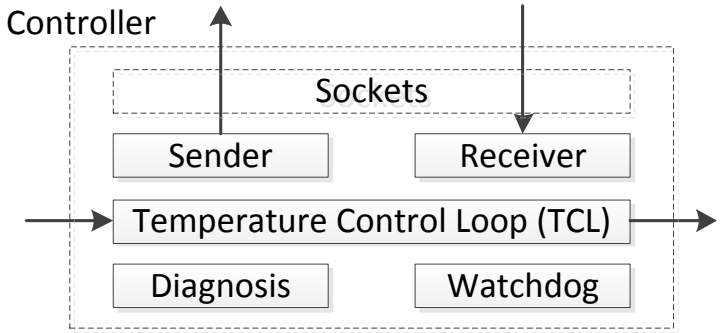

Figure 3.1: Components of the controller

first, starts the five tasks, and then waits for the finish of the tasks, as the control-flow diagram shows in Figure 3.1. The TCL task reads temperature data from I/O ports connecting to the sensors (through A/D converters), computes the changes of the temperature and adjusts the flow rate of coolants, and sends outputs to I/O ports connecting to the actuators (through D/A converters). If the temperature exceeds a preset limit, a trip is issued to stop the reaction (and sound the alarm).

The sender task sends to the control center the status of the controller stored in a shared buffer, which is protected by a pthread mutex. When a task wants to update the controller status, it enables the corresponding status bit in the buffer. Once sent, the shared buffer is cleared by the sender. The receiver task receives control commands (for example, shutdown the controller gracefully) from the control center through the network, which is a FDDI network virtually structured as two rings. When the receiver receives control commands, it execute the commands directly. In order to prevent the receiver from waiting incoming data forever, the receiver may times out when listening data from the network. Both receiver and sender rely on sockets to receive and send data.

The diagnosis task performs several hardware tests, for example network interfaces failure, physical memory faults, I/O ports errors, to make sure that all the hardware of the controller works correctly. The watchdog task resets the hardware watchdog timer after checking periodically if the other tasks are still alive and progressive. Except the watchdog task, each task is required to increment a counter after finishing its job in each round. A task is considered dead if its counter is not incremented in time.

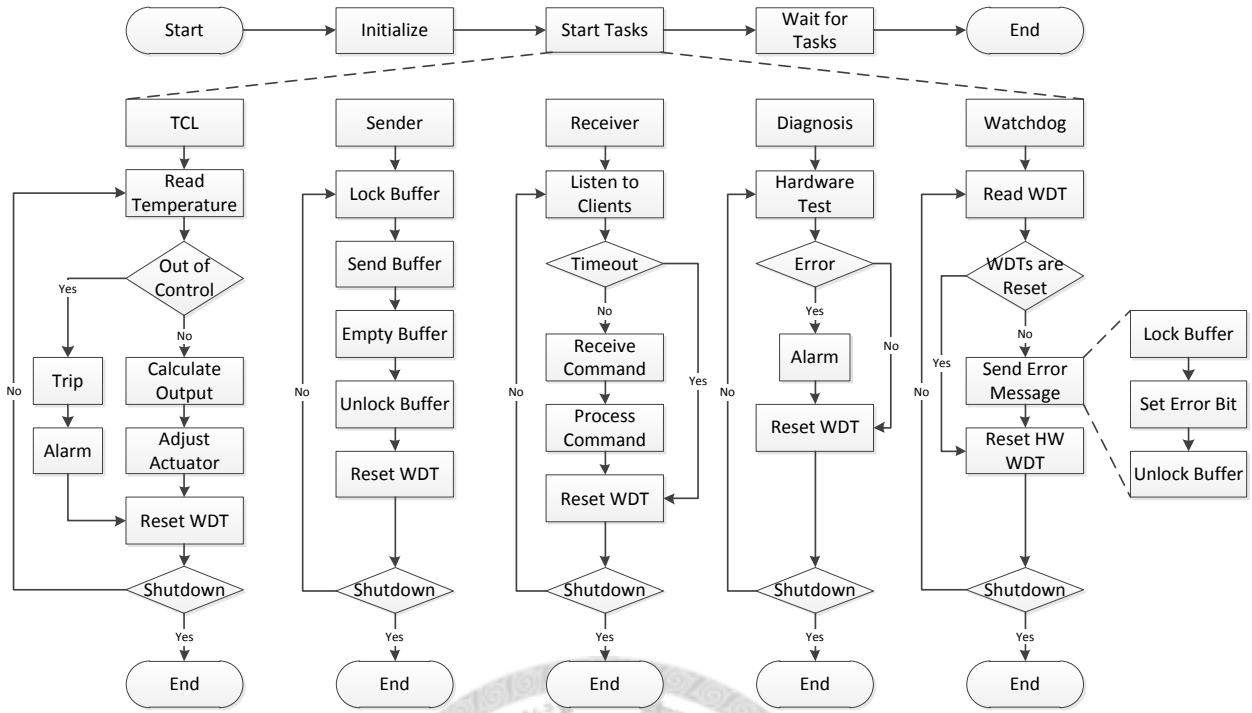

Figure 3.2: Control flows of the controller threads

3.2 Requirements

Each thread/task of the controller should correctly complete its jobs in a timely manner. The requirements for the controller and the verification tasks that they entail can be grouped into the following three categories; Figure 3.3 illustrates this grouping.

- *Basic program analysis.* The controller should continue to run without crashing for simple program safety reasons or getting deadlocked. So, its program should be statically analyzed to ensure that it is free of any common program safety error such as out of bound array index, buffer overflow,illegal pointer access, and memory leakage. It should also be analyzed to ensure that there will be no deadlock. These basic program safety analyses may be performed with static program analyzers such as Astrée and CodeSonar. Since they are essentially push-button processes, we will not elaborate on them further in this paper.

- *Functional correctness.* The main task here is to verify that the output of the TCL task in each round is indeed 1 (to increase the flow rate of coolants) when the most recent four sampled temperature readings are increasing and the last reading

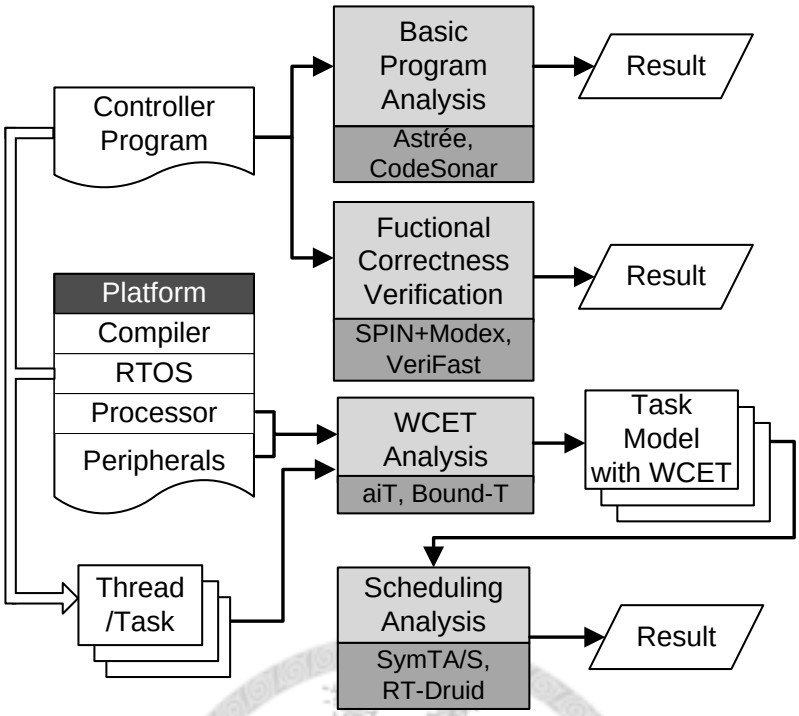

Figure 3.3: Relevant verification tasks for the controller

exceeds a set high temperature value, the output should be -1 (to decrease the flow rate) when the most recent four sampled temperature readings are decreasing and the last reading is under a set lower temperature value, the output should be 0 (to maintain the flow rate), otherwise. Also, the watchdog task correctly detects the inactive of other tasks and there is no false alarm. This relies on that the counter of a task is incremented at the end of each round, though quite straightforward, should be verified. Any discovered error will be reported eventually. In Chapter 4, we show how these correctness requirements can be formalized and verified with the SPIN model checker.

- *Timing correctness* (WCET and scheduling analyses). The main timing requirements for the controller are: (1) the TCL, Receiver, and Watchdog tasks must each complete one round of its jobs in every 25 milliseconds, (2) the Sender task in every 50 milliseconds, and (3) the Diagnosis task in every 8 hours. This involves a safe estimation of the WCET of each thread/task and a scheduling analysis to see if the

jobs of all tasks for one round will be completed before their respective deadline, i.e., end of the corresponding period. Note that these analyses require information about the execution platform, including the compiler, RTOS, processor, and peripherals. In Chapter 5, we show how these analyses may be carried out with the aiT and SymTA/S tools.

3.3 Code of the Controller

Listing 3.1: The main thread

```
1  int main(void)
2  {
3    pthread_t tcl_t, sender_t, receiver_t, diagnosis_t, watchdog_t;
4    pthread_attr_t attr;
5    struct sched_param param;
6    int errno;
7
8    /* Initialize the five tasks. */
9    if (errno = initialize())
10     return errno;
11
12   /* Initialize the base time. */
13   gettime(&starttime);
14
15   pthread_attr_init(&attr);
16
17   param.sched_priority = TCL_PRIORITY;
18   pthread_attr_setschedparam(&attr, &param);
19   if (errno = pthread_create(&tcl_t, &attr, tcl, NULL))
20     return errno;
21
22   param.sched_priority = SENDER_PRIORITY;
23   pthread_attr_setschedparam(&attr, &param);
24   if (errno = pthread_create(&sender_t, &attr, sender, NULL))
25     return errno;
26
27   param.sched_priority = RECEIVER_PRIORITY;
28   pthread_attr_setschedparam(&attr, &param);
29   if (errno = pthread_create(&receiver_t, &attr, receiver, NULL))
30     return errno;
31
32   param.sched_priority = DIAGNOSIS_PRIORITY;
33   pthread_attr_setschedparam(&attr, &param);
34   if (errno = pthread_create(&diagnosis_t, &attr, diagnosis, NULL))
35     return errno;
36
37   param.sched_priority = WATCHDOG_PRIORITY;
38   pthread_attr_setschedparam(&attr, &param);
39   if (errno = pthread_create(&watchdog_t, &attr, watchdog, NULL))
40     return errno;
```

```
41
42    pthread_join(tcl_t, NULL);
43    pthread_join(sender_t, NULL);
44    pthread_join(receiver_t, NULL);
45    pthread_join(diagnosis_t, NULL);
46    pthread_join(watchdog_t, NULL);
47
48    finalize();
49
50    return EXIT_SUCCESS;
51  }
```

Listing 3.2: The TCL task

```
1   void* tcl(void* arg)
2   {
3     int pos, neg, prev_temp, curr_temp, temp1, temp2;
4     int size = 4;
5
6     waitForNext(TCL_PERIOD);
7     while(state == STATE_RUNNING) {
8       prev_temp = curr_temp;
9
10      read(sensor1, &temp1, sizeof(temp1));
11      read(sensor2, &temp2, sizeof(temp2));
12      curr_temp = (temp1 + temp2) / 2;
13
14      if(curr_temp > prev_temp){
15        if(pos == 3)
16          pos = pos;
17        else
18          pos = pos +1;
19      }
20      else
21        pos = 0;
22      if(curr_temp < prev_temp){
23        if(neg == 3)
24          neg = neg;
25        else
26          neg = neg +1;
27      }
28      else
29        neg = 0;
30
31      if (curr_temp > MAX_TEMPERATURE) {
32        /* Stop the reaction. */
33        trip();
34        alarm(TCL, ERR_TEMPERATURE);
35      } if (pos >= 3 && curr_temp > HIGH_TEMPERATURE) {
36        /* Increase the flow rate of coolants. */
37        increase(actuator);
38      } else if (neg >= 3 && curr_temp < LOW_TEMPERATURE) {
39        /* Decrease the flow rate of coolants. */
40        decrease(actuator);
41      }
```

```
42
43      wdt_tcl = (wdt_tcl + 1) % MAX_WDT;
44
45      waitForNext(TCL_PERIOD);
46    }
47  }
```

Listing 3.3: The sender task

```
1  void* sender(void* arg)
2  {
3    waitForNext(SENDER_PERIOD);
4    while(state == STATE_RUNNING) {
5      pthread_mutex_lock(&mutex);
6      write(s_sockfd, buffer, strlen(buffer));
7      bzero(buffer, BUFFER_SIZE);
8      pthread_mutex_unlock(&mutex);
9
10     wdt_sender = (wdt_sender + 1) % MAX_WDT;
11
12     waitForNext(SENDER_PERIOD);
13   }
14 }
15
16 void enable(int module, char flags)
17 {
18   pthread_mutex_lock(&mutex);
19   buffer[module] = buffer[module] & flags;
20   pthread_mutex_unlock(&mutex);
21 }
```

Listing 3.4: The receiver task

```
1  void* receiver(void* arg)
2  {
3    int nbytes;
4    char buffer[BUFFER_SIZE];
5    int retval;
6
7    waitForNext(RECEIVER_PERIOD);
8    while(state == STATE_RUNNING) {
9      /* Check if incoming data are available with a timeout. */
10     retval = waitOnSocket(r_sockfd, 1);
11     if (retval == -1)
12       error(RECEIVER, ERR_RECEIVER_SOCKET_FAILED);
13     else if (retval == 1) {
14       if ((cli_sockfd = accept(r_sockfd, (struct sockaddr *) &
             cli_addr, sizeof(cli_addr))) == -1) {
15         error(RECEIVER, ERR_RECEIVER_SOCKET_FAILED);
16       } else {
17         bzero(buffer, BUFFER_SIZE);
18         nbytes = read(cli_sockfd, buffer, BUFFER_SIZE - 1);
19         close(cli_sockfd);
```

```
20          process(buffer);
21        }
22      }
23
24      wdt_receiver = (wdt_receiver + 1) % MAX_WDT;
25
26      waitForNext(RECEIVER_PERIOD);
27    }
28  }
```

Listing 3.5: The diagnosis task

```
1  void* diagnosis(void* arg)
2  {
3    int round = 0;
4    char errors;
5
6    waitForNext(DIAGNOSIS_PERIOD);
7    while(state == STATE_RUNNING) {
8      errors = 0;
9
10     /* Check sensors. */
11     if (round == 0) {
12       if (isSensorDown())
13         errors |= ERR_SENSOR_DOWN;
14     }
15
16     /* Check actuators. */
17     if (round == 1) {
18       if (isActuatorDown())
19         errors |= ERR_ACTUATOR_DOWN;
20     }
21
22     /* Check network connection. */
23     if (round == 2) {
24       if (isNetworkDown())
25         errors |= ERR_INTERFACE_DOWN;
26     }
27
28     if (errors)
29       error(DIAGNOSIS, errors);
30
31     round = ++round % 3;
32     wdt_diagnosis = (wdt_diagnosis + 1) % MAX_WDT;
33
34     waitForNext(DIAGNOSIS_PERIOD);
35   }
```

Listing 3.6: The watchdog task

```
1  void* watchdog(void* arg)
2  {
3    unsigned char tcl[2] = {0, 0};
```

```
4   unsigned char sender[2] = {0, 0};
5   unsigned char receiver[2] = {0, 0};
6   unsigned char diagnosis[2] = {0, 0};
7   unsigned int tcl_r = 0;
8   unsigned int sender_r = 0;
9   unsigned int receiver_r = 0;
10  unsigned int diagnosis_r = 0;
11  unsigned int tcl_i = 0;
12  unsigned int sender_i = 0;
13  unsigned int receiver_i = 0;
14  unsigned int diagnosis_i = 0;
15  char errors;
16
17  waitForNext(WATCHDOG_PERIOD);
18  while(state == STATE_RUNNING) {
19    errors = 0;
20
21    if (tcl_r == TCL_WDT_ROUNDS) {
22      tcl_i = (tcl_i + 1) % 2;
23      tcl[tcl_i] = wdt_tcl;
24      if (tcl[0] == tcl[1])
25        errors |= ERR_TCL_WDT;
26      tcl_r = 0;
27    }
28    else tcl_r++;
29
30    if (sender_r == SENDER_WDT_ROUNDS) {
31      sender_i = (sender_i + 1) % 2;
32      sender[sender_i] = wdt_sender;
33      if (sender[0] == sender[1])
34        errors |= ERR_SENDER_WDT;
35      sender_r = 0;
36    }
37    else sender_r++;
38
39    if (receiver_r == RECEIVER_WDT_ROUNDS) {
40      receiver_i = (receiver_i + 1) % 2;
41      receiver[receiver_i] = wdt_receiver;
42      if (receiver[0] == receiver[1])
43        errors |= ERR_RECEIVER_WDT;
44      receiver_r = 0;
45    }
46    else receiver_r++;
47
48    if (diagnosis_r == DIAGNOSIS_WDT_ROUNDS) {
49      diagnosis_i = (diagnosis_i + 1) % 2;
50      diagnosis[diagnosis_i] = wdt_diagnosis;
51      if (diagnosis[0] == diagnosis[1])
52        errors |= ERR_DIAGNOSIS_WDT;
53      diagnosis_r = 0;
54    }
55    else diagnosis_r++;
56
57    if (errors)
58      error(WATCHDOG, errors);
```

```
59
60      /* Reset the hardware watchdog timer. */
61      resetWDT();
62
63      waitForNext(WATCHDOG_PERIOD);
64    }
65  }
```

Listing 3.7: Helper functions

```
1  void waitForNext(unsigned long period)
2  {
3    unsigned long d, s;
4    timespec_t ts;
5
6    /* Calculate the elapsed time. */
7    gettime(&ts);
8    d = diff(&ts, &starttime);
9
10   /* Sleep until the next round. */
11   s = period - (d % period);
12   if (s > 0)
13     msleep(s);
14 }
15
16 void error(int module, char flags)
17 {
18   display(module, flags);
19   enable(module, flags);
20 }
21
22 void enable(int module, char flags)
23 {
24   pthread_mutex_lock(&mutex);
25   buffer[module] = buffer[module] & flags;
26   pthread_mutex_unlock(&mutex);
27 }
28
29 int waitOnSocket(int fd, int ms)
30 {
31   fd_set rfds;
32   struct timeval tv;
33
34   FD_ZERO(&rfds);
35   FD_SET(fd, &rfds);
36   tv.tv_sec = 0;
37   tv.tv_usec = ms;
38
39   return select(fd+1, &rfds, NULL, NULL, &tv);
40 }
41
42 int isNetworkDown()
43 {
44   struct ifaddrs *itfs = NULL, *itf = NULL;
45   int d = 0;
```

```
46
47    if (getifaddrs(&itfs) == 0) {
48      for (itf = itfs; itf != NULL; itf = itf->ifa_next) {
49        int down = itf->ifa_flags & IFF_UP == 0 || itf->ifa_flags &
              IFF_RUNNING == 0;
50
51        if (strcmp(itf->ifa_name, INTERFACE) == 0 && down)
52          d = 1;
53      }
54    }
55    freeifaddrs(itfs);
56    itfs= NULL;
57
58    return d;
59  }
```

Chapter 4

Functional Correctness

Functional correctness of a program can be verified by two main approaches, model checking and deductive approach. In this section, we demonstrate the verification steps of the controller with the state-of-the-art model checker SPIN and the deductive verifier VeriFast.

4.1 Model Checking - Using SPIN

The model checking approach searches automatically in an abstract model to see if user-specified assertions are violated. Below we show how we verify the tasks of the controller program with the model checker SPIN and the model extractor Modex.

4.1.1 Define Modex test harness file

A model of SPIN is specified in PROMELA (PROcess MEta LAnguage). To mechanically extract the model of the controller program with Modex, we need to define a Modex test harness file, which is a file with the extension *.prx*. Below are the typical steps of defining a Modex test harness file.

1. Define the target program name.
2. Define the target procedures name.
3. Define the filters.
4. Define how the model pieces are combined and specify the LTL formulae.
5. Add some declarations to satisfy Modex parser in the C-source file.

We will use the above steps to verify two of the main tasks in verifying the controller program. One is in the sender thread, which has mutual exclusion issue. The other is in the computation thread, which has some required properties.

- Mutual Exclusion Task

  There is a shared buffer that will be accessed from five threads in the controller. Each thread will write the corresponding bit in the buffer when errors occur. The sender thread will send out and reset the buffer every period. The controller program uses *mutex* to prevent two threads write or send the buffer at the same time. The following shows how we verify the mutual exclusion issue.

  1. Define the target program name.

     For defining the target single C-source file, we use command `%F`.

     ```
     %F sender.c
     ```

  2. Define the target procedures name.

     For defining the target procedures name in the C-source file, we use command `%X`. And for defining which filter the procedure will used, we add option `%L`. Also, for defining the extract model of procedure as an active prototype, we add option `%a`.

     The sender thread sends the status stored in a shared buffer, which is protected by a pthread mutex. Other threads access the buffer through the enable function in `sender.c`. Therefore, we choose the sender and enable functions to be the target procedures.

     ```
     %X -L sender -a sender
     %X -a enable
     ```

  3. Define the filters.

     For defining the details of the filters, we use command `%L`. It is a multi-line command, and `%%` is to specified the end of the definition. If `%L` is followed by a name, it will apply to the specified procedure, and if not, it will apply to all procedures.

```
%L
Import  _all_   _all_
pthread_mutex_lock(&(mutex))     atomic { mutex == 0 ->
   mutex = 1 }; assert(mutex==1)
pthread_mutex_unlock(&(mutex))   mutex = 0
%%
%L sender
bzero(...         hide
waitForNext(...   hide
%%
```

Above are two filters: the first applies to all procedures, and the second only applies to the sender procedure. The `Import` command specifies how the data objects are included in the generating model. The first argument of it is to define the name of a data object, and the second argument is to define the scope in which it appears. Arguments are separated with the tab key. In this task we use the keyword `_all_` to express all data objects in all scopes (local and global) are included in the model.

Other lines specify the mapping we want from code to model. The first argument is to match the expression in code, and the second argument is to define how the expression will appear in the model. Because the functions `bzero` and `waitForNext` are not concerned with the mutual exclusion issue we care, we use the keyword `hide` to hide the two functions in the model. The parameters of the function are not needed to fully specify, Modex will match the expression with the string before three dots. To model the pthread function with *mutex*, we map the lock function to an atomic sequence, which checks if the *mutex* is free (equals to zero), locks the *mutex* (assign one to the *mutex*). Moreover, to ensure that *mutex* will be one after the locking function, we add the assertion `assert(mutex==1)` in the model. Similarly, the mapping of unlock function is to assign zero to the *mutex*.

4. Define how the model pieces are combined and specify the LTL formulae.

   For defining how various pieces of model are combined and executed, we use command `%P`, which is also a multi-line command. The extra declarations are defined here when they are needed to model the behaviors of the source code. Also, we can specify LTL formulae here to check the required properties.

   ```
   %P
   ```

```
#include "_modex_sender.spn"
#include "_modex_enable.spn"
ltl live { [](mutex==1)-><>(mutex==0)) }
%%
```

Because the sender and enable procedures do not have other behaviors that need to be modeled, we simply include the two procedures to model with `#include` command. And to ensure that the *mutex* will always eventually be unlocked after being locked, we specified the LTL formula `ltl live []((mutex==1)-><>(mutex==0))`. *ltl* is the command to show that its a LTL formula. *live* is the name of the formula, and when verifying the model, we can choose which formula to verify through the name.

5. Add some declarations to satisfy Modex parser in the C-source file.

   Sometimes we need more declarations to help Modex to parse the code. In this task, the data type of *mutext* is declared to be *pthread_mutex_t* in the source code, but because Modex does not recognize the data type *pthread_mutex_t*, parse error will occur. We thus add declaration to define that the data type of *pthread_mutex_t* is integer in the source file.

```
#ifdef MODEX
    #define pthread_mutex_t int
    #define pthread_t int
    #define timespec_t int
#endif
```

- Required Properties Task

  The computation thread required to satisfy the following property: when the most recent four sampled temperature readings are increasing and the last reading exceeds a set high temperature value, the computation thread will call *increase* function to increase the flow rate of coolants. The following shows how to verify the property.

1. Define the target program name.

```
%F computation.c
```

2. Define the target procedures name.

```
%X -a tcl
```

3. Define the filters.

```
%L
Import  _all_   _all_
waitForNext(...  hide
read(sensor1...  read_in1 ? temp1
read(sensor2...  read_in2 ? temp2
trip()   break
increase(...     increase_flag=1;
decrease(...     decrease_flag=1;
alarm(...        hide
%%
```

The aims of two *read* functions in source code are to read the temperatures from sensor1 and sensor2. Because the temperatures may be varied each time, we should additional model the behavior. We map the *read* functions to two channels, which will get temperatures from two procedures that we will define in 4). The aim of the *trip* function is to stop the reaction, and thus we map it to the keyword *break* to break the model. The aim of the *increase* function is to increase the flow rate of coolants, and therefore we assign one to the variable *increase_flag* in the model, showing that the *increase* function has been called.

4. Define how the model pieces are combined and specify the LTL formulae.

```
1  %P
2  int increase_flag=0;
3  int decrease_flag=0;
4  chan read_in1 = [1] of { int };
5  chan read_in2 = [1] of { int };
6
7  #include "_modex_tcl.spn"
8
9  active proctype read1()
10 {
11   int rand_temp=0;
12   do
13   ::
14     if
15     :: rand_temp<=300;
16       rand_temp=rand_temp+1;
17     :: rand_temp>=-60;
18       rand_temp=rand_temp-1;
19     fi;
20     read_in1 ! rand_temp;
21   od;
22 }
23
24 active proctype read2()
25 {
```

```
26    int rand_temp=0;
27    do
28    ::
29      if
30      :: rand_temp<=300;
31        rand_temp=rand_temp+1;
32      :: rand_temp>=-60;
33        rand_temp=rand_temp-1;
34      fi;
35      read_in2 ! rand_temp;
36    od;
37  }
38
39  #define p0 (tcl:pos == 0)
40  #define p1 (tcl:pos == 1)
41  #define p2 (tcl:pos == 2)
42  #define p3a (tcl:pos == 3)
43  #define p3b (tcl:pos == 3 && tcl:curr_temp > 180 && tcl:
        curr_temp <= 300)
44  #define inc (increase_flag == 1)
45
46  ltl live { [](( p0 U p1 U p2 U p3a U p3b) -> <>(p3b U inc))
        }
47  ltl safe { [](<>inc -> (p0 U p1 U p2 U p3a U p3b)) }
48  %%
```

The first four lines declare the variables that do not appear in the source code but in the model. *increase_flag* and *decrease_flag* are variables to detect whether the *increase* or *decrease* functions are being called. *read_in*1 and *read_in*2 are two channels that can store one message of type *int*. Using `#include "_modex_tcl.spn"` to include the *tcl* procedure.

*read*1 and *read*2 are two procedures to model the temperature sensors. We use non-deterministic choices to simulate the variation of the temperatures. Each iteration in *read*1 will either increase or decrease the temperature and send it to *tcl* procedure through *read_in*1 channel. The procedure *read*2 does similarly.

The first LTL formula

```
ltl live { [](( p0 U p1 U p2 U p3a U p3b) -> <>(p3b U inc))
    }
```

is to ensure when the most recent four sampled temperature readings are increasing, and the last reading exceeds the set high temperature value, the increase function will be called. Using *p*3*a* and *p*3*b* to differentiate the states that the temperature has continuously increased three times but does not

exceed the set high temperature value from the temperature do exceed the set high temperature at the last increasing sampled.

The second formula

```
ltl safe { [](<>inc -> (p0 U p1 U p2 U p3a U p3b)) }
```

is to ensure when the increase function has been called, there must happened that four sampled temperature readings are continuous increasing, and the forth exceeds the set high temperature value.

Note if the LTL formula is

```
ltl safe { [](<>inc -> (p0 U p1 U p2 U p3b)) }
```

, SPIN will report an error for not satisfying the property. Because it miss the trace that the temperature has continuously increased three times but does not exceed the set high temperature value, and then the temperature keeps continuously increased until it exceeds. The trace unsatisfied `(p0 U p1 U p2 U p3b)`, but it also calls the increase function. Therefore the error occurs.

5. Add some declarations to satisfy Modex parser in the C-source file.

   This task do not need additional declarations.

4.1.2 Run Modex to Extract Model

After defining the *.prx* file, we run Modex to extract the model. Use command `modex filename.prx`, for example:

```
modex sender.prx
```

It will automatically extract the model and save in *_modex_.drv*, and then use the following command to pass it to *model*.

```
cpp -E -P _modex_.drv > model
```

For preprocessing but not compiling the source file, we add option $-E$. And, for inhibiting the generation of line-control information, we add option $-P$.

4.1.3 Run SPIN to Verify the Model

After the extraction of the model, we run SPIN to verify it with the following command.

```
spin -O -a model
```

The option $-O$ is to turn off the new scope rules, and $-a$ is to generate a verifier for the specification. The output of the command are pan.[cbhmt] files. Then use gcc compiler to produce an executable verifier.

```
gcc -DMEMLIM=1024 -o pan pan.c
```

The option $-DMEMLIM$ is to define the memory size in Megabytes that can be allocated, and $-o$ is to place output in file *pan*. If the verification runs out of memory, add the option $-DCOLLAPSE$ to compress state vectors. And if the memory is still not insufficient, change the compression option to $-DBITSTATE$, which is an approximate bit-state hashing or supertrace technique [35] [34].

In our required properties task, we countered the insufficient memory problem. Because the supertrace technique may have approximation errors, we first lowered the required scope and did the exhaustive verification. We changed the set high temperature value from 180 to 4, and maximum temperature value from 300 to 7. With the above changes, it can be exhaustive verified, and then we did supertrace verification. Through the steps, although the original source is still not exhaustive verified, the result of supertrace verification is reasonably reliable.

Besides, if the target property is a safety property, add the option $-DSAFETY$ to increase the search efficiency.

Finally, run *pan* to show the result. For verifying safety property:

```
./pan -m10000000 -N safe
```

Use option $-mN$ to set max search depth to N steps. If the specified LTL formulae are more than one, use option $-N$ and follow the name of LTL formula that want to verify.

For verifying liveness property:

```
./pan -m10000000 -a -f -N live
```

Use option $-a$ to find acceptance cycles, and $-f$ to add weak fairness.

After the execution of above commands, if there are errors in verification result, use the following command to trace the error trail.

```
spin -t -c model
```

The option $-t$ is to perform a guided simulation, and $-c$ is to display the message sequence chart for a guided run.

4.2 Deductive Approach - Using VeriFast

Typical verification steps before invoking a deductive verifier include rewriting the program, modeling external functions, and annotating the program. In the following, we illustrate the three steps with the deductive verifier VeriFast [38].

4.2.1 Rewrite Program

The first restriction of VeriFast is the limitted support of global variables, for example global variables are not allowed in loop invariants. A solution is putting all global variables in a big C struct passed as an argument to all threads. The second restriction is on-stack variables of non-primitive types. Unlike the forbidden heap usage in MISRA-C, VeriFast requires that all C structs and arrays must be created in the heap by `malloc` and deleted by `free`. Another unsupported program feature is single writer and multiple readers of an atomic variable. An `int` variable on a POSIX system with GNU C libraries can be assumed atomic [5] and thus it is unnecessary to protect the variable with mutex or locks if it has only one single writer. The watchdog counters of the controller are exactly atomic `int` variables. In VeriFast, a thread always have some permission f on a heap trunk. The permission f is a real number between 0 (excluded) and 1 (included). If a thread has 1 permission (default if not specified) on a heap trunk, it can read and write on the heap trunk. Otherwise, the thread can only read the heap cell. When a thread allocates a heap trunk, the thread gets full permission on the heap trunk. Such a permission model cannot support the single writer and multiple readers of an atomic variable because otherwise the full permission will be greater than 1. Thus we have to protect these atomic variables explicitly in VeriFast. There are other restrictions not mentioned in this paper.

4.2.2 Model External Functions

For pthreads, VeriFast provides a set of annotated functions with different names and signatures from the standard pthread interfaces. For other unsupported system calls and libraries, instead of providing their source code, we can model their semantics by giving their specifications in pre-conditions and post-conditions. For example, below shows a specification of the system call `open`.

```
/*@ predicate on(int flags, int flag) = true == ((flags & flag) ==
    flag); @*/

/*@ predicate opened(int fd); @*/

int open(char *pathname, int flags);
  /*@ requires [?f]chars(pathname, ?cs) &*& mem('\0', cs) == true
     &*& on(flags,
  O_RDONLY | O_WDONLY | O_RDWR); @*/
  /*@ ensures result >= -1 &*& (result == -1 ? true : opened(result)
     ); @*/
```

The VeriFast-style annotations appear as block comments enclosed by `/*@` and `@*/` or line comments starts with `//@`. The notation `&*&` denotes separating conjunction in separation logic [49]. The precondition requires that `pathname` points to a null-terminated string, the current thread has f permission ($f > 0$) on the string, and `flags` includes one of the access modes: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. Since the `open` operation does not modify the pathname, it only needs some non-negative permission for read access. The post-condition ensures that the returned code is -1 on error. The predicate `opened(result)` indicates that the file descriptor `result` has been associated with an opened file. To ensure that a `close` operation must follow a successful `open` operation in every execution path, we specify the `close` operation as a consumer of an `opened` predicate.

```
int close(int fd);
//@ requires opened(fd);
//@ ensures true;
```

In VeriFast, a predicate is treated as a heap trunk and thus it will report memory leaks if a file descriptor is opened but not closed later in an execution path.

The thread synchronization with mutex is already modeled in VeriFast. The specifications of mutex functions are shown below.

```
struct mutex *create_mutex();
//@ requires create_mutex_ghost_arg(?p) &*& p();
//@ ensures mutex(result, p);

void mutex_acquire(struct mutex *mutex);
//@ requires [?f]mutex(mutex, ?p);
//@ ensures mutex_held(mutex, p, currentThread, f) &*& p();

void mutex_release(struct mutex *mutex);
//@ requires mutex_held(mutex, ?p, currentThread, ?f) &*& p();
//@ ensures [f]mutex(mutex, p);

void mutex_dispose(struct mutex *mutex);
//@ requires mutex(mutex, ?p);
//@ ensures p();
```

Basically, the predicate `p` represents the invariant of the resource protected by a mutex, denoted by `m` in the following. The creation of the mutex requires the presence of the invariant and protects the invariant in a predicate `mutex(m, p)`, which can be divided into several copies each having permission less than 1 and owned by a thread. The invariant is guaranteed once a thread successfully acqures the mutex and has to be reestablished before releasing the mutex.

4.2.3 Annotate Program

Consider the shared buffer of the sender task as an example. A basic invariant of the shared buffer is:

```
g->buffer |-> ?b &*& g->buffer_size |-> ?s &*& chars(b, ?cs) &*&
    malloc_block(b, s) &*& length(cs) == s
```

where `g` is the struct containing all global variables, `s->f |-> ?v` indicates that some value `v` is stored in the field `f` of a struct `s`, `b` points to an allocated heap blocks `cs` of length `s` and of type `char` is ensured by `chars(b, ?cs) &*& malloc_block(b, s) &*& length(cs) == s`. Adding more predicates to the invariant can make the sender task know more about the status of the buffer. For example, a safety property may require that whenever the sender task sends the buffer with the `i`-th error bit enabled, the corresponding error `err` must happen sometime before. Such property can be specified by the following formula and added to the invariant of the shared buffer.

```
nth(i, cs) != 1 || err == 1
```

However, without support of temporal properties in VeriFast, a similar liveness property cannot be specified: whenever the error `err` happens, the buffer sent by the sender task will eventually have the corresponding error bit `i` enabled.

Consider the function `waitOnSocket`, which takes a file descriptor of a socket connection and a timeout value, waits for incoming data from the socket connection until the timeout expires, and returns the results of the waiting. The return value may be 0 on timeout or -1 on error. The specification of the function `waitOnSocket` is shown below.

```
int waitOnSocket(int fd, int timeout)
  /*@ requires fd >= 0 && timeout >= 0; @*/
  /*@ ensures result == 0 || result == -1; @*/
```

The requires clause specifies the precondition of the function while the ensures clause specifies the postcondition of the function.

Loop invariant is a key point in the deductive approach. VeriFast does not infer sufficient loop invariants automatically, and thus loop invariants should be specified explicitly. For example, consider the receiver task which relys on a server socket connection `sockfd` to accept client's connection. This server socket connection should not be changed while running. To specify this property, ghost variables will be needed.

```
//@ int SOCKFD = sockfd;
while(STATE == STATE_RUNNING)
/*@ invariant sockfd = SOCKFD; @*/
{
  ...
}
```

Before entering the receiver's periodic task in the while loop, a ghost variable `SOCKFD` is created in the annotation and the value of `sockfd` is assigned to `SOCKFD`. The loop invariant then specifies after each round of the while loop, the server connection socket `sockfd` is still equal to `SOCKFD`, which is never changed.

For checking the absense of buffer overflow and pointer errors, verification conditions are generated by VeriFast automatically. A drawback is that the integer bound is fixed in VeriFast and thus is not platform dependent.

Consider the following correctness property specific to the controller: whenever there are three consecutive temperature raises and the last temperature is higher than $HIGH\_TEMPERATURE$, the coolant flow rate increases. The formal description of this property is stated in the following. Let t_i be the i-th sampling time point ($t_i < t_j$ iff $i < j$), $temp[t]$ the average temperature measured at time point t, and $cool[t]$ the coolant flow rate decided at time point t. For all four consecutive sampling time points t_i, t_{i+1}, t_{i+2}, and t_{i+3}, if $temp[t_i] < temp[t_{i+1}] < temp[t_{i+2}] < temp[t_{i+3}]$ and $temp[t_{i+3}] > HIGH\_TEMPERATURE$, then $cool[t_{i+3}] > cool[t_{i+2}]$. However, the controller does not keep four most recent temperature history. Instead, it keeps the previous measured temperature and the count of consecutive temperature raises *pos* before the current time point. The count *pos* is incremented if the current temperature is higher than the previous temperature and is reset otherwise. Another problem is that existing verifiers usually support variable reference at a specific line of code, for example $x@l$ refers to the value

of x at label l, but not reference to a variable at a specific loop iteration, for example $temp[t_i]$.

As a workaround, we can introduce $temp[t_i]$, $temp[t_{i+1}]$, $temp[t_{i+2}]$, and $temp[t_{i+3}]$ as ghost variables to the program, prove the desired property, and also prove the relation between the ghost variables and the count pos. Below illustrates how these ghost variables are inserted and how the relation is specified.

```
1  int size = 4;
2
3  /*@
4   predicate is_zero(int e) = e == 0;
5
6    lemma int* create_ghost(int size);
7      requires size > 0;
8      ensures ints(result, size, ?is) &*& foreach(is, is_zero);
9
10   lemma void set_ghost(int *arr, int idx, int value);
11     requires ints(arr, ?size, ?is1) &*& idx >= 0 &*& idx < size;
12     ensures ints(arr, size, ?is2) &*& is2 == update(idx, value, is1)
           ;
13
14   fixpoint bool consecutive(list<int> is, int size, int curr, int
       idx, nat pos) {
15     switch(pos) {
16       case zero: return curr == (idx-1)%size ||
17                  nth(idx, is) <= nth((idx-1)%size, is);
18       case succ(n): return nth(idx, is) > nth((idx-1)%size, is) ?
             consecutive(is, size, curr, idx-1, n) : false ;
19     }
20   }
21 @*/
22
23 //@ int prev = 0;
24 //@ int curr = 0;
25 //@ int *temp = create_ghost(size);
26 while(STATE == STATE_RUNNING)
27 /*@ invariant ints(temp, size, ?is) &*& 0 <=
28                prev &*& prev < size &*& 0 <=
29                curr &*& curr < size @*/;
30 {
31     //@ prev = curr;
32     //@ curr = (curr + 1) % size;
33     prev_temp = curr_temp;
34     //@ set_ghost(temp, prev, curr_temp);
35
36     ...
37
38     curr_temp = temp1 + temp2;
39     //@ set_ghost(temp, curr, curr_temp);
40
41     ...
42
```

```
43      if (pos >= 3 && curr_temp > HIGH_TEMPERATURE) {
44      //@ assert consecutive(is, size, curr, curr, nat_of_int(pos))==
            true;
45        increase(actuator);
46      }
47      ...
48  }
```

The lemma `create_ghost` is used to create an array of ghost variables `temp` that tracks four of the most recent temperature measurements. The lemma `set_ghost` is used to update elements in the temperature array. The recursive predicate `consecutive` is used to relate the array of temperature and the count `pos` to make sure that the calculation of `pos` is correct.

Chapter 5

Timing Correctness

To garantee timing correctness, we first calculate the WCET of each thread, and then using the WCET to caculate the WCRT of the controller program, and finally add the effect of overhead to the WCRT result. We assume using aiT as the WCET tool, and SymTA/S as the WCRT tool.

5.1 Compute WCET

Computing WCET of each thread includes dividing and compiling the program, annotating the executable binaries, and running the tool.

5.1.1 Divide and Compile the Program

To treat individual tasks separatetly, we divide the target program into six parts, each part from an independent thread, which are the main, TCL, receiver, watchdog, sender, and diagnosis.

Each piece of code should contain all the information that the task needs. For example, the header files included and the global functions that each thread calls all need to be included. In the controller program, the part of the main thread code includes the header of the main function, and the main function itself. And the threads that use error function to send error message should include the error function in the TCL task, and the `enable` function that the `error` function calls in the sender task.

Except the main thread, all the other threads use the buffer as a shared variable. The locking mechanism in the contoller program assure that there will only one thread access the buffer at a time. The locking instruction is non-preemptive and the others

are preemptive. Therefore, different scheduling algorithm be applied in the next section, and thus we need divide each thread to three parts. The WCET of each part is needed for scheduling analysis. The codes of first part is from the start of while loop to the instruction before locking. The second part includes locking instructions, and the third part includes the codes after locking instruction in the while loop. For example, the first part of the TCL task in the while loop is as follows.

```
prev_temp = curr_temp;
...
if (curr_temp > MAX_TEMPERATURE) {
    trip();
}
```

The second part is

```
alarm(ERR_TEMPERATURE);
```

And the third part is:

```
if (pos >= 3 && curr_temp > HIGH_TEMPERATURE) {
    increase(actuator);
}
...
waitForNext(TCL_PERIOD);
```

The receiver thread calls the locking function `error` in both of the branches.

```
if (retval == -1)
    error(ERR_RECEIVER_SOCKET_FAILED);
else if (retval == 1) {
    if ((cli_sockfd = accept(r_sockfd, (struct sockaddr *) &cli_addr
        , sizeof(cli_addr))) == -1) {
         error(ERR_RECEIVER_SOCKET_FAILED);
    ...
```

Acutully, it will only execute one branch of the codes according to the condition. Thus, we first calculate the WCET of the whole receiver thread, and the result will show which branch has longer execution time. Then, using annotation to set the condition true for the longer branch, and divide the thread into three parts. For example, if we find the "if" branch has larger WCET, the first part is as follows.

```
retval = waitOnSocket(r_sockfd, 1);
if (retval == -1)
```

The second part:

```
error(ERR_RECEIVER_SOCKET_FAILED);
```

And the third part:

```
else if (retval == 1) {
...
waitForNext(RECEIVER_PERIOD);
```

After the separation, compile each part of the code to generate executable binary files. In our example, we use GCC to compile.

5.1.2 Annotate Program

We assume to use aiT as the WCET tool, which analyzes executable binaries. The information of the executable is usually not sufficient to compute accurate WCET bounds, therefore we need to use annotations to provide further information. For example, annotation of the compiler can help aiT to reconstruct the control flow better.

```
/* ai: compiler "arm-gcc"; */
```

The receiver thread is periodic with the while loop. Due to that the code outside of the while loop is not periodic, and our focus is on the timing behavior that threads periodic occur, we calculate the WCET of periodic part only. As our goal is to verify that all of the tasks will complete one round within 50 milliseconds, we use annotation to compute the WCET in one cycle.

```
while(STATE == STATE_RUNNING) {
    /* ai: loop here begin exactly 1; */
    ...
    waitForNext(RECEIVER_PERIOD);
  }
```

After computing the WCET of the whole receiver thread, as the preview section mentioned, we use annotation to set the condition for the branch that with larger WCET to be true. For example, if we find the "if" branch has larger WCET, then we set the variable `retval` to be -1 to calculate the WCET of the first part of the thread.

```
if (retval == -1)
    /* ai: instruction here is entered with retval=-1; */
    ...
```

Also, we assign the value of registers that aiT can not determine from value analysis. For example, in the diagnosis thread we need to assign the number of the interface, which varys from systems.

```
for (interface = interfaces; interface != NULL; interface =
   interface->ifa_next) {
    /* ai: instruction here is entered with interfaces=7; */
```

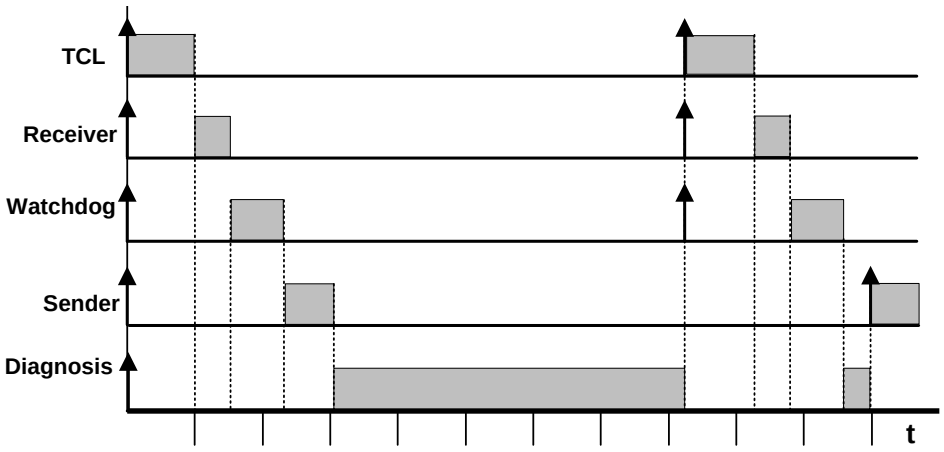

Figure 5.1: Scheduling of five periodic threads.

The above annotation represents that the value of interfaces will always be seven at the address.

5.2 Compute WCRT

After calculating the WCET of each thread, we use scheduling tool to analyze the WCRT of the controller program. The steps include setting up system model, quering for WCET, and running the tool.

5.2.1 Set up System Model

There are three different periods of tasks in the controller program. The diagnosis thread has the longest period, and all the other threads except the sender have the same period, and the period of the sender thread is twice as much as them. The higher priority thread in the controller program has shorter period, and thus it can apply in rate-monotonic scheduling [46] [45]. The main thread is excluded for it only executes once. The priorities of the threads are as follows, TCL = receiver = watchdog > sender > diagnosis. Figure 5.1 shows a possible execution sequence without jitter. The gray blocks represent the execution time of the thread in a period, and the up-arrows indicate the activating time of each period.

Fugure 5.2 shows the system model of our example. The grey squares represent the input source, i.e. input event model, and the white squares represent the task model. The

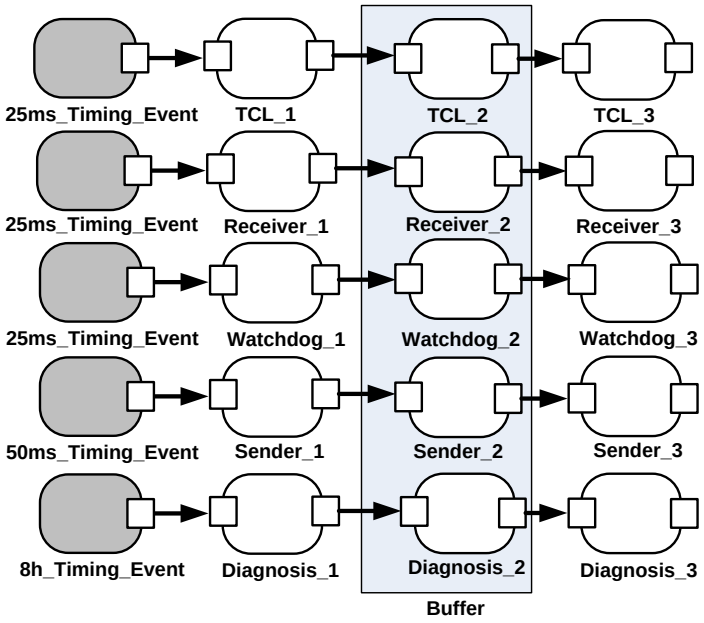

Figure 5.2: System model of the controller program. It is illustration not screen shot from the tool.

five threads are triggered by different periodic timing events. Due to locking of the shared buffer, it may cause priority inversion in the locking instructions part, i.e., a lower priority thread can block all other high-priory threads once it enters the locking area. Thus we divide each thread into three parts, tasks with "_1" is the first part of thread, it represents the part before locking, and task with "_3" is the part after locking. Both of the two parts are preemptive, and we choose rate monotonic as scheduling method. Task with "_2" is the locking part, and the gray square which surrounds all the locking parts indicates that all of the threads share the same buffer. The locking part is non-preemptive, and we choose fixed priority non-preemptive scheduling.

5.2.2 Query for WCET and Run

SymTA/S supports integration with aiT. In the user interface of SymTA/S, it can run WCET directly. As the system model we present, the second part will do in each iteration. Actually, the system is designed to write buffer when an error occurs. However, we can not tell when will error happen, so the worst situation is every error occurs in every iteration. If we can verify that the model satisfies the timing constraints in the worst situation, the program will always satisfy the timing constraints.

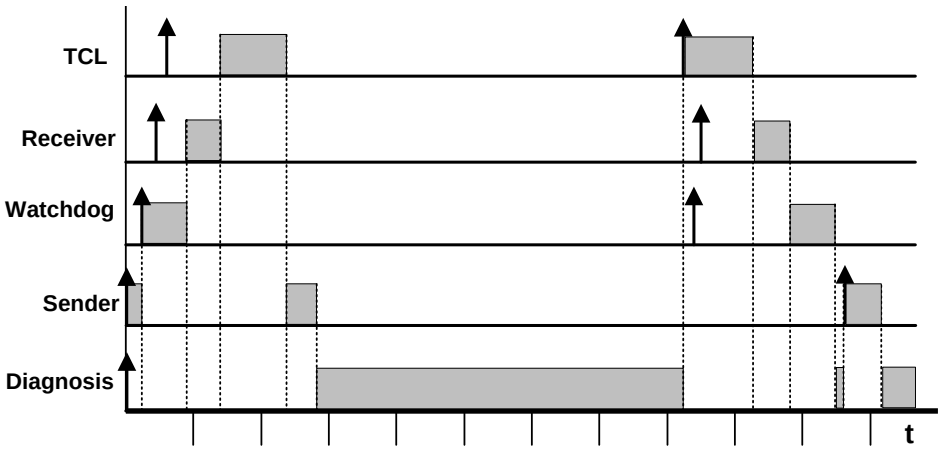

Figure 5.3: Scheduling of five periodic threads with jitters.

5.3 Compute Overhead

There will be overhead due to context-switch, shared cache, memory interference, etc. Therefore the actual threads are periodic with jitters, the preemption may occur when the lower priority threads execute first. Figure 5.3 shows an example that preemption occurs, if the activations of TCL, receiver and watchdog thread delay for jitter, the sender thread will execute first, but when the thread with higher priority activates, it will be preempted.

When using aiT and SymTA/S, below are the typical steps to compute task switch cost.

1. Query task switch cost. In user interface of SymTA/S, we both query the WCET and task switch cost for every individual tasks we divided from aiT.

2. Run scheduling analysis.

5.3.1 Query Task Switch Cost

To consider the context-switch overhead, we use the UCB (Useful Cache Block) analysis provided in aiT to compute task switch costs. The value of UCB will be stored in XTC (XML Timing Cookie) [10], which is an interchange format that both of the tools can communicate through.

5.3.2 Run Scheduling Analysis

The UCB will add to all the next tasks with higher priority as termination overhead, i.e., the WCRT of task will add termination overhead of the lower priority tasks that be preempted. The result of WCRT with overhead can be viewed as numerical or Gantt charts.

Chapter 6

Conclusion

It is challenging to make multithreaded programs right because of concurrency issues. For instance, synchronization may introduce errors in timing-dependent data, and also mutually-exclusive operations are needed to prevent shared data race. Real-time programs are required to guarantee strict timing constraints. However, multithreaded programs are difficult to debug and trace execution paths, and real-time programs with multiple threads have more complex timing behaviors. Therefore, real-time multithreaded programs are prone to mistakes during programming, and it is desirable to apply formal verification on such programs.

In this thesis, we reviewed methods and tools for verifying real-time multithreaded programs, and we considered a representative controller program as our target program to provide a more comprehensive illustration. We assembled a selection of tools to completely verify the functional and timing correctness of a controller program. Also, we described the issues and requirements of a real-time multithreaded program, and showed the details about how to perform the verification tasks. Moreover, we pointed out that several verification tasks still have to be carried out by hand, and there also exist gaps between the model for verification and the real program.

6.1 Contributions

- **Guidance on the usage of tools and tasks for verification of real-time multithreaded programs.**

  We select the verification tools to be a complete tool chain for functional and timing

correctness, and use the tools to verify the representative controller program. The thesis can guide the practitioner to attain the high assurance guaranteed by formal methods.

- **Point out the tasks that still have to be carried out by hand, and there are gaps between abstract model and program.**

  Verification in some of tools is not fully automatic, and transformation may be needed to handle the verification process. Besides, some of the verification tasks are not supported by current tools, and also there may exist gaps between the abstract model and the real program. Finding out whether these gaps exist can remind the practitioner to notice the gaps, and to notice the influences that the gaps may lead to. We also hope that our study will encourage tool developers to try filling the gaps.

6.2 Discussion

6.2.1 SPIN+Modex

Model extraction is nearly fully automatically with Modex. Although the reliability of the mechanically transformation without proving, the correctness of mechanically transform part of model is highly trusted. Below we discuss the reasons that may cause differences between the abstract model and the source code.

First, the parts that need manually transform to model. Programs usually send and receive messages from the outside environment, and if the messages are varied with time, we should manually model the changing behavior as in our required properties task. Actually, we should model the behaviors that will change the state while the program is running and the model checker can not know how it is changed. The manually part effects much of the correctness of the model, and it needs to comprehend the interactions between the program and other program or environment.

Second, there are some of forms of code can not be transformed with Modex. e.g `condition ? valueIfTrue : valueIfFalse`. Modex will have parse error with above argument, but we can simply change it to the `if-else` form. There may be other code

forms that are not supported by Modex, but if we can change the form without modifying the meaning, there will not generate gaps that from the changing of code forms.

Third, there are difficulties in modeling function calls. If there are two threads both call the function F, Modex will transform the threads to two procedures, and transform the function call to an extended active prototype, which is instrumented to support procedure call mechanism. However, F is only one process, and it can not be reentered, which is against the meaning in source code. We may change to map F as inline to solve the problem, but if F has parameters or return value, it does not work.

6.2.2 VeriFast

We have tried to verify the controller for the same tasks with VeriFast. And we find there are more gaps between the abstract model and the source code when using VeriFast. One of the reasons is because there are many features that do not support. For example, global variables can not use in VeriFast. However, there are common to use global variables in programs, so we need to rewrite greatly for solving the features that do not support.

Moreover, it need to model all of the external functions such as system functions and third-party libraries. We need to specify the pre-conditions and post-conditions for all the functions that the program called, and this need comprehension of the details of the external functions. If the pre/post-condition are not well-defined, it will generate the gaps between the rewrite code to the original code.

Although VeriFast is prone to generate the gaps than SPIN, when facing programs that with deep hierarchies of function calls, it will need much manually transformation with SPIN. Thus the rewrite program for VeriFast may be closer to the source code, and the verification with VeriFast may be more reliable. Also, if the annotations are accurate, the semantic of program will be more faithful with VeriFast.

6.3 Future Work

- Complete the verification of timing correctness of the controller program with tools.

  In the thesis, because of the difficulties in acquiring of the tools and the hardware,

the timing correctness of the controller do not actually verify with the tools. The experience of actually verifying the timing correctness can help recognize more about the execution of the tools.

- Complete the verification of functional correctness with VeriFast, and find out in what situations it has more benefits than model checking method.

  We have pointed out some of the advantages and disadvantages of VeriFast, and if the controller can actually be verified with it, the differences between VeriFast and SPIN will be more obvious. And the experience may suggest the programmers to choose tool according to the target tasks.

- Increase the number of issues that need to be verified in the controller program.

  There may be more issues that can be discussed in the controller program. And the experience can suggest the programmer what verification issues may a controller program face.

- Include real-time issue in SPIN.

  SPIN now can not deal with real-time problems, but there are some methods have been proposed. Including the real-time issues in SPIN may simpler the tool chain of verification, or give some timing information to the tool for timing correctness.

- Apply more tools on the verification, and compare the results, discuss the advantages and disadvantages.

  There are many verification tools, comparing the the advantages and disadvantages can suggest the programmers to choose the suitable tools for verification.

Bibliography

[1] aiT. http://www.absint.com/ait/.

[2] Bound-T. http://www.bound-t.com/.

[3] DO-178B/ED-12B. Software Considerations in Airborne Sstems and Equipment Certification.

[4] Frama-C software analyzers. http://frama-c.com/.

[5] The GNU C library manual.

[6] IEC 61508. http://www.iec.ch/functionalsafety/.

[7] IEEE Std 1003.1-2008 Portable Operating System Interface (POSIX) Base Specifications, Issue 7.

[8] IEEE Std 1003.1-2008 Portable Operating System Interface (POSIX) Commands and Utilities, Issue 7.

[9] IEEE Std 1003.1-2008 Portable Operating System Interface (POSIX) System Interfaces and Headers, Issue 7.

[10] INTERESTED project.

[11] Modex. http://spinroot.com/modex/index.html.

[12] RT-Druid. http://www.evidence.eu.com/content/view/28/51/.

[13] SWEET. http://www.mrtc.mdh.se/projects/wcet/sweet.html.

[14] SymTA/S. http://www.symtavision.com/symtas.html.

[15] MISRA-C: 2004 — guidelines for the use of the C language in critical systems, 2004.

[16] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[17] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV 2011*, volume 6806 of *LNCS*, pages 184–190, 2011.

[18] S. Byhlin, Ermedahl A., Gustafsson J., and Lisper B. Applying static wcet analysis to automotive communication software. In *ECRTS 2005*, pages 249–258, 2005.

[19] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV 2002*, volume 2404 of *LNCS*, pages 359–364, 2002.

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169, 2000.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.

[22] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: a practical system for verifying concurrent c. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 23–42, 2009.

[23] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 331–340, 2011.

[24] C. Cuypers, B. Jacobs, and F. Piessens. Verification of data-race-freedom of a java chat server with verifast. CW reports CW550, Department of Computer Science, K.U.Leuven, 2009.

[25] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[26] B. Dutertre and L. d. Moura. The YICES SMT solver.

[27] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. In *Proceedings of the 1st International Workshop on Embedded Software*, volume 2211 of *LNCS*, pages 469–485, 2001.

[28] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV 2007*, volume 4590 of *LNCS*, pages 173–177, 2007.

[29] P. R. Glück and G. J. Holzmann. Using spin model checking for flight software verification. In *Proceedings of Aerospace Conference, 2002. IEEE*, volume 1, pages 105–113, 2002.

[30] J. Gustafsson and A. Ermedahl. Experiences from applying WCET analysis in industrial settings. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, pages 382–392. IEEE Computer Society, 2007.

[31] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH, 2004. http://www.absint.com/wcet.htm.

[32] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. *IEE Computers and Digital Techniques*, 152(2):148–166, 2005.

[33] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[34] G. J. Holzmann. An improved protocol reachability analysis technique. *SOFTWARE, PRACTICE AND EXPERIENCE*, 18:137–161, 1988.

[35] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.

[36] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 2003.

[37] Md. I. Hossain and N. S. Chowdhury. A practical approach on model checking with modex and spin. In *International Journal of Electrical and Computer Sciences*, volume 11, pages 1–7, 2011.

[38] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*, volume 6461 of *LNCS*, pages 304–311, 2010.

[39] M. Jersak. Compositional performance analysis for complex embedded applications. Technical report, PhD thesis, Technical University of Braunschweig, 2004.

[40] D. Kästner, R. Wilhelm, R. Heckmann, M. Schlickling, M. Pister, M. Jersak, K. Richter, and C. Ferdinand. Timing validation of automotive software. In *ISoLA*, pages 93–107, 2008.

[41] M. Kim, S. Hong, C. Hong, and T. Kim. Model-based kernel testing for concurrency bugs through counter example replay. *Electronic Notes in Theoretical Computer Science*, 253(2):21–36, 2009.

[42] J. Kosina. Fighting security bugs in the linux kernel. In *WDS'07 Proceedings of Contributed Papers*, pages 64–71, 2007.

[43] S. Künzli, A. Hamann, R. Ernst, and L. Thiele. Combined approach to system level performance analysis of embedded systems. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 63–68, 2007.

[44] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47:700–713, 1998.

[45] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm exact characterization and average case behavior. In *Proceedings of Real Time Systems Symposium 1989*, pages 166–171, Santa Monica, CA, USA, 1989.

[46] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2002.

[47] L. d. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340, 2008.

[48] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. 2002.

[49] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society Press, 2002.

[50] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proceedings of the 2nd World Congress on Formal Methods*, volume 5850 of *LNCS*, pages 532–546, 2009.

[51] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.3*, October 2010.

[52] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[53] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36–1, 2008.

[54] J. Yoo, E. Jee, and S. Cha. Formal modeling and verification of safety-critical software. *Software, IEEE*, 26(3):42–49, 2009.

[55] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software*, volume 5156 of *LNCS*, pages 325–342, 2008.