

Ph.D. DISSERTATION

Formal Verification Framework for Cyber-Physical Systems on PALSware

사이버 물리 시스템을 위한
PALSware 시스템 엄밀 검증 프레임워크

BY

Yoonseung Kim

AUGUST 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Formal Verification Framework for Cyber-Physical Systems on PALSware

사이버 물리 시스템을 위한
PALSware 시스템 엄밀 검증 프레임워크

BY

Yoonseung Kim

AUGUST 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY


Formal Verification Framework for Cyber-Physical Systems on PALSware


지도 교수 허 충 길


이 논문을 공학박사 학위논문으로 제출함
2021 년 6 월

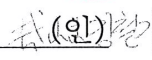
서울대학교 대학원
전기.컴퓨터공학부
김 윤 승


김윤승의 공학박사 학위논문을 인준함
2021 년 6 월

위 원 장 _____ 이광근 

부위원장 _____ 허충길 

위 원 _____ 이창건 

위 원 _____ 김철기 

위 원 _____ 강지훈 

Abstract

Achieving high-level safety guarantees for cyber-physical systems has always been a key challenge, since many of those systems are safety-critical so that their failures in the actual operation may bring catastrophic results. Many cyber-physical systems have real-time and distributed features, which increase the complexity of the system an order of magnitude higher.

In order to tame the complexity, a middleware called PALSware has been proposed. It provides a logically synchronous environment to the application layer on top of physically asynchronous underlying network and operating systems. The complexity of a system can be significantly reduced in a synchronous environment.

However, a bug in PALSware may have destructive effects since it exposes every application system to runtime failures. Moreover, finding bugs in PALSware can be very challenging in some cases, for various reasons.

To solve this problem, we present VeriPALS, a formally verified C implementation of PALSware together with a verification framework for application systems. Especially, the framework provides an executable model as an efficient random testing tool. As case studies, we developed two application systems, and applied VeriPALS to them in order to demonstrate effectiveness of the framework in both testing and formal verification.

Keywords: formal verification, distributed systems, real-time systems, synchronous systems, theorem proving

Student Number: 2013-23107

Contents

Abstract	i
Chapter 1 Introduction	1
Chapter 2 Preliminaries	8
2.1 PALSware	8
2.1.1 PALSware in a Distributed System	9
2.1.2 Correctness of Synchronization on Reliable Network	10
2.1.3 Implementation of PALSware	11
2.2 Interaction Trees	14
Chapter 3 Overview	16
3.1 Framework	16
3.2 Key Ideas	21
3.2.1 Concurrent Executions of Nodes	21
3.2.2 Global Clock vs. Local Clock	22
3.2.3 Real-time Local Executions of Node Model	23
3.2.4 Time Constraint on Network Transmission Times	24
3.2.5 Time Constraint on Program Executions	25

3.2.6	Observable Behaviors of a Real-Time Distributed System . . .	26
Chapter 4	Formalization	28
4.1	General Definitions	28
4.2	Application System of the Framework	31
4.3	Real-World Model	34
4.3.1	Network Model	34
4.3.2	Generic System Model on Network	35
4.3.3	Operating System Model	37
4.4	Executable Abstract Synchronous Model	41
4.5	Result	42
Chapter 5	Refinement Proof using Intermediate Models	44
5.1	Refinement 1: Abstraction of C programs	44
5.2	Refinement 2: Abstract PALSware	47
5.3	Refinement 3: Abstraction of Network	48
5.4	Refinement 4: Synchronous Execution	51
5.5	Refinement 5: Making It Executable	54
Chapter 6	Case Study 1: Active-Standby Resource Scheduling System	55
6.1	High-Level Description	56
6.2	Implementation	59
6.3	Formally Verified Properties	62
6.3.1	Correctness of Implementation	62
6.3.2	Abstraction to Single-Controller System	63
Chapter 7	Case Study 2: Synchronous Work Assignment System	68
7.1	High-Level Description	69
7.2	Implementation	70

Chapter 8 Results	75
8.1 Development	75
8.2 Experimental Results	77
Chapter 9 Related Work	80
9.1 PALS Pattern and PALSware Verification	80
9.2 Verification Frameworks for Distributed Systems	81
9.3 Verifying C Programs	83
Chapter 10 Conclusion and Future Work	85
Bibliography	88
초록	92
Acknowledgements	93

List of Figures

Figure 1.1	Comparison between asynchronous and synchronous systems .	2
Figure 1.2	Comparison between the asynchronous and synchronous active-standby systems	3
Figure 2.1	A distributed system built on PALSware	9
Figure 2.2	An example of logically synchronous executions on PALSware	10
Figure 2.3	Structure of PALSware implementation	12
Figure 2.4	Simplified definition of interaction tree in Coq	14
Figure 2.5	An example interaction tree, the exact definition and a monatic-style representation	15
Figure 3.1	The overall structure of the verification framework	18
Figure 4.1	Formal semantics of the network model	36
Figure 4.2	Formal semantics of generic systems on network	36
Figure 4.3	Transition steps of process	38
Figure 4.4	Formal semantics of the operating system model	40
Figure 4.5	Executable abstract synchronous model	41
Figure 5.1	Program simulation proof in Ref. 1	46

Figure 5.2	AbsPALS	47
Figure 5.3	Semantics of abstract asynchronous model	49
Figure 5.4	Ref 3. Abstract asynchronous model	50
Figure 5.5	Semantics of synchronous model	52
Figure 5.6	Semantics of synchronous nodes	53
Figure 5.7	Ref 4. Multi-step simulation between asynchronous model and synchronous model	53
Figure 6.1	An execution of the active-standby system	58
Figure 6.2	C implementation of the console task	59
Figure 6.3	C implementation of the controller tasks	60
Figure 6.4	C implementation of the device tasks	65
Figure 6.5	The abstract specification for the console task	66
Figure 6.6	The abstract specification for the controller tasks	66
Figure 6.7	The abstract specification for the device tasks	67
Figure 7.1	An execution of the work assignment system	71
Figure 7.2	C implementation of the master task	73
Figure 7.3	C implementation of the worker tasks	74
Figure 8.1	Experimental results comparing performance of the three test- ing methods	78

List of Tables

Table 8.1	Lines of code in the C development	76
Table 8.2	Lines of code in the Coq development	77

Chapter 1

Introduction

Achieving high-level safety guarantees for cyber-physical systems has always been a key challenge of the field, since many of those systems are safety-critical, such as autonomous vehicle systems, avionics systems, and nuclear systems[1, 2, 3]. For those systems, a failure occurred during the actual operation may bring catastrophic results, as in actual cases of a power plant and aerospace systems[4, 5, 6]. However, many cyber-physical systems have both real-time and distributed features, which increase the complexity of the system an order of magnitude higher. In the process of real-time communication between distributed components of the system, the randomness of program execution times, clock skews, transmission times, and many other factors all together may lead to a combinatorial explosion of possible executions, and it is difficult to correctly implement and verify the system with consideration of all the possibilities.

In order to tame the complexity of real-time distributed systems, a middleware, called PALSware, has been proposed[7]. PALSware provides a logically synchronous environment to the application layer on top of physically asynchronous underlying

networks (with real-time operating systems), by appropriately controlling the message deliveries and the job execution timings.

To illustrate how the complexity of a system can be significantly reduced in a synchronous environment, we use a simple example shown in Fig. 1.1. The system is composed of three electronic control units (ECU) connected by network, where each ECU runs its own tasks P , Q , R , and S on a real-time operating system, as shown in Fig. 1.1(a). The flow of the execution is as follows. First, on ECU 1, P sends the message a to ECU 2, and then sends the message b to ECU 3. Then, responding to the messages, ECU 2 and ECU 3 launches Q and S respectively. Finally, S on ECU 3 sends the message c to ECU 2, that triggers the execution of R .

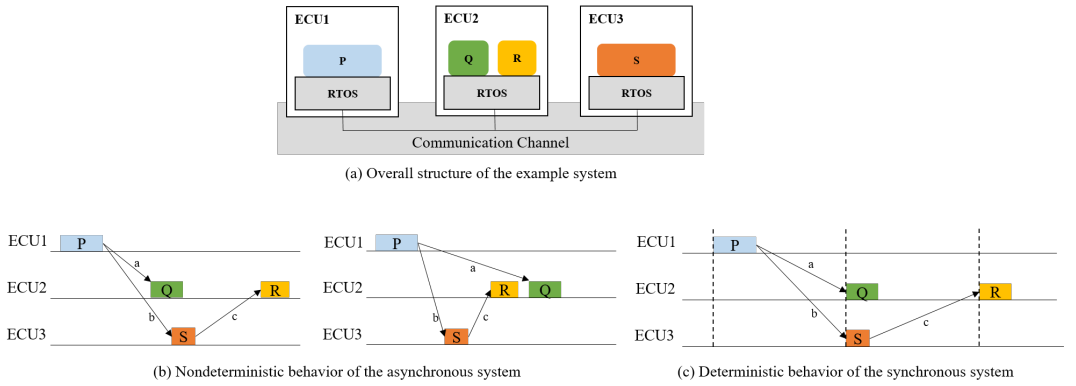


Figure 1.1 Comparison between asynchronous and synchronous systems

An asynchronous implementation of the example system may behave nondeterministically, so that the two different executions depicted in Fig. 1.1(b) are made possible. In the first execution, the message a arrives at ECU 2 earlier than the message c from S , that makes Q run before R . However, in the second execution, the order of message arrivals to ECU 2 are reversed, and as a consequence, R runs before Q .

On the other hand, a synchronous implementation may reduce the nondetermin-

ism in the system, as shown in Figure 1(c). With synchronization, the three ECUs run tasks simultaneously according to the designated period, and resulting messages are delivered to the next round of execution, without any race condition on the arrival times of messages. This deterministic property makes it easier to predict the behavior of the system, which greatly helps to correctly implement and verify the system.

The active-standby system, used as a motivating example for PALSware in [9, 10, 7], clearly shows the complexity of asynchronous system designs. The system consists of three tasks: two controller tasks that are replicated for fault-tolerance, and one console task that receives input from the user. During execution, one of the controllers is supposed to be in the active mode, and the other one in the standby mode. The two controllers regularly send “heartbeat” messages that contains their status to each other, so that the standby side can detect if the active side fails, and if it happens, the stanby side switches its status to the active mode. Also, the user can give input to the console to toggle the active side, and then the console sends the “toggle” message to each controller.

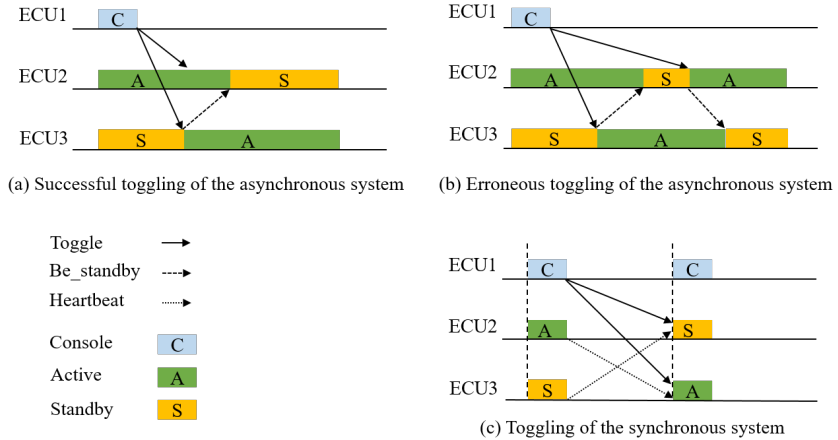


Figure 1.2 Comparison between the asynchronous and synchronous active-standby systems

A defective asynchronous design of the active-standby system described in Fig. 1.2(a) and (b) shows the difficulty of constructing correct asynchronous systems. In the design, the toggle functionality works as follows: when the console sends a “Toggle” message, only the standby side responds to the message by switching the status and sending a “Be_standby” message to the active side. The active side just ignores the Toggle message, to prevent toggling when the standby side has failed. After receiving Be_standby, the former active side switches its status to the standby mode. A successful toggling execution is depicted in Fig. 1.2(a). However, an erroneous case happens if the Toggle message arrives at the former active side after it already switched its status as depicted in Fig. 1.2(b). Then, it initiates another toggle that effectively cancels the first toggle. The situation gets more complicated if we introduce heartbeat messages to handle node failures.

In contrast, a synchronous design of the active-standby system enables the system to operate under a simpler logic, as shown in Fig. 1.2(c). Suppose that the console sends a Toggle message in a period. Then, at the beginning of the next period, each controller checks whether the Toggle message and the heartbeat message from the other side have arrived, and if so, the controllers switch their status. Note that this design doesn’t require another type of messages such as Be_standby. Now that the system becomes simpler, it is easier to reason about the correctness of the system. In conclusion, implementing a real-time distributed system on PALSware may greatly reduce the complexity of the system.

While PALSware alleviates the complexity problem of application systems, there remains a question about the correctness of PALSware itself. An implementation or design bug in PALSware may have destructive effects since it exposes every application system built on top of the PALSware to danger of runtime failures. However, finding all such bugs with testing may be hard for the following reasons. First, due to various nondeterministic factors such as message communication and job execution

timings, such a bug in PALSware may appear only in certain corner cases that are hard to detect and reproduce. Second, since PALSware should work correctly for any application systems, we should test it for all such applications, which is simply impossible. Therefore, PALSware is a good target for formal verification, which would provably guarantee the absence of bugs.

In this paper, we present *VeriPALS*, a formally verified C implementation of PALSware together with a verification framework for application systems building on top of it. Our contributions are summarized as follows.

- We implemented a (slightly simplified) version of PALSware in C and formally proved in Coq that it is correct with respect to (*i.e.*, behaviorally refines) an executable abstract synchronous model that provides a logically synchronous environment to application systems. For this verification, we developed a proof method that allows us to gradually prove refinement between a physical asynchronous model and an abstract synchronous model via multiple intermediate models. In particular, the models and proof method allow us to model and reason about clock skew among distributed components and also properly model assumptions about worst case execution time (WCET) of C code, which is a part of our trust base (*i.e.*, we suppose it will be separately verified by other WCET analysis tools).
- The abstract model supports both efficient testing and formal verification of application systems on top of it. Specifically, testing on the abstract model greatly improves efficiency in time and resources over testing on the physical system because the abstract system does not physically wait for the next period to come, and does not use the physical network either. Also any formal verification about an application system on top of the abstract model (*e.g.*, proving certain safety or liveness properties) is automatically transformed to that on

top of the physical model by combining it with the correctness proof of the PALSware implementation.

- As case studies, we developed two application systems built on PALSware, and applied VeriPALS to the systems for testing and verification. The first system is a more general and practical version of the above active-standby system, which contains two replicated controllers that work together as a resource scheduling task, and several devices that occasionally request a resource. Using VeriPALS, we first performed random testing for the system on top of the abstract model, and then formally verified in Coq the property that the system with two controllers works equivalently as that with a single never-failing controller under the assumption that both controllers do not fail at the same time. The second system is a synchronous work assignment system, which contains a master task which assigns works generated in real time to multiple worker tasks. For the second case study, we only applied random testing for cost-effective verification.

In order to formally specify the behavior of the PALSware and PALSware application code written in C, we use the Clight formal semantics from the CompCert project[12]. The Clight semantics covers large subset of the C language, which also serves as the language semantics for a number of prominent C-program verification projects[13, 14, 15]. For the future work, we plan to combine the verification result of the CompCert compiler with ours, to guarantee that the compiled assembly code of the PALSware and application code still preserves the synchronous behavior.

The rest of the paper is structured as follows. In Chapter 2, we introduce background knowledge for reading. Specifically, it explains PALSware and interaction trees. Chapter 3 gives a high-level overview of the framework. We give brief explanation of the structure of framework, and list several key ideas. Next, in Chapter 4, we present the formalization of the framework. It contains formal system models

and the final theorem. Then, in Chapter 5, we describe the sketch of the refinement proof for the final theorem. We explain the intermediate model and refinement proof for each step of the whole proof. Chapter 6 and 7 presents our case studies. We develop two application systems, and applied the framework to them, for testing and formal verification. Chapter 8 contains the volume information of our development and experimental results about testing. Chapter 9 introduces several related studies and compares them with our work. The last chapter concludes and suggests future research directions.

Chapter 2

Preliminaries

This chapter introduces background knowledge that is helpful for understanding the content of this paper. First, we focus on how PALSware functions to establish a synchronous environment for the upper application layer. Also, we introduce interaction trees[16], which we use to express abstract, yet executable, specifications in the framework.

2.1 PALSware

PALSware is a middleware that implements the physically-asynchronous logically-synchronous (PALS) architectural pattern[9] in order to provide a virtual synchronous environment to the application layer. We first give a high-level overview of the role of PALSware in a distributed system, and then explain how PALSware correctly synchronizes all distributed tasks under several assumptions on the underlying physical environment. Finally, we give the structure of our implementation of PALSware, and explain simplified aspects in our version.

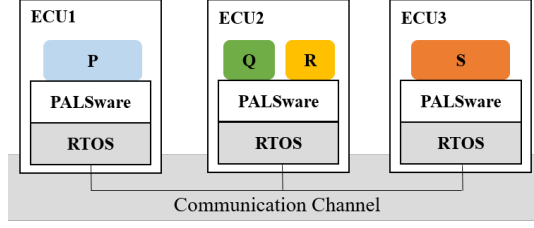


Figure 2.1 A distributed system built on PALSware

2.1.1 PALSware in a Distributed System

The overall structure of a distributed system built on PALSware is depicted in the example of Fig. 2.1. In the figure, there are three *nodes*, which are ECU 1, 2, and 3, connected to the communication channel that serves as the network. Each node consists of three layers. On the ground layer, a real-time operating system manages hardware devices and offers system call services to the upper layer. In the middle, PALSware provides a synchronous environment to the top layer. In particular, PALSware uses the network socket and timer services of the operating system. On the top layer, an *app* program runs on PALSware. We call the collection of apps in a system an *application system* of PALSware. Also, we say an app implements a *task* of the distributed system. In this perspective, *P* and *S* in the example are tasks, while *Q* and *R* are regarded as subroutines that comprise a task on ECU 2. Each task is assigned a unique nonnegative number as its task ID.

At runtime, PALSware periodically executes its task for every *synchronization period* of the system. Each instance of the periodic execution is called a *job* of the task. Through PALSware, a task may send a message to either a single task or multiple tasks using the multicast service of network routers. PALSware uses the UDP protocol for message communication. Even though it lacks the reliability provided by TCP, the *reliable network assumption* that we are going to explain in the following section justifies the use of UDP.

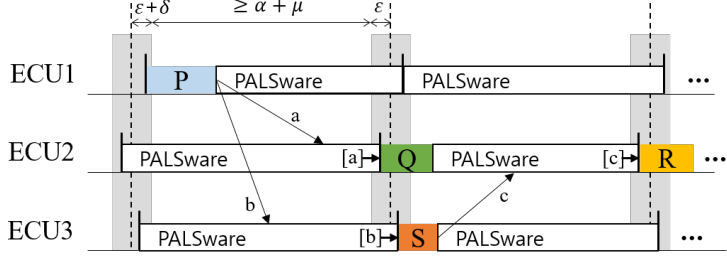


Figure 2.2 An example of logically synchronous executions on PALSware

2.1.2 Correctness of Synchronization on Reliable Network

The synchronization process of PALSware relies on several assumptions on the underlying physical environment. First of all, it works on the reliable network assumption: an end-to-end message delivery through the network is guaranteed to be done within a fixed maximum transmission time μ , without duplication or loss of the message. Another assumption is that the real-time operating system guarantees that the local clock skew is bounded by the maximum clock skew ϵ , which can be achieved by using one of the known clock synchronization algorithms[17]. Also, it assumes that the sleep functionality of the real-time operating system guarantees to wake up the program within the maximum local-time delay δ from the appointed wake-up time. The first two assumptions are given in the existing work, where the last assumption is introduced by us, to reason about the concrete implementation of PALSware.

Under those assumptions, PALSware logically synchronizes the distributed tasks with two main operations: (1) running the task at the beginning of each period, and (2) keeping messages arrived from other tasks and then pass them to the task at the right destination period.

Let's see how PALSware works with the previous simple example, as illustrated in Fig. 2.2. First, PALSware at ECU 1 runs the task P at the beginning of the first period, where the actual time could be at most $\epsilon + \delta$ away from the exact time due to

the clock skew and the wake-up delay. During the execution of P , the two messages a and b are sent to ECU 2 and ECU 3 respectively, attached with the sender's task ID and the destination time, which is the next period. After a certain amount of transmission time less than μ , each message arrives at its destination. Since messages are sent before completing the sender's job execution, the arrival times of the messages are earlier than $\alpha + \mu$ after the sender begins the job. When the next period starts, PALSSware on ECU 2 runs Q with the message a , and PALSSware on ECU 3 runs S with the message b which sends the message c to ECU 2. Finally, at the third period, PALSSware at ECU 2 runs R with the message c .

The essential condition for the correct synchronization is that the length of the period should be long enough so that all messages sent in one period are guaranteed to arrive before the receiving tasks of the next period start. Considering the maximum job execution time α , the period T should satisfy the condition below, as shown in Fig. 2.2.

$$T \geq \alpha + \mu + 2\varepsilon + \delta$$

2.1.3 Implementation of PALSSware

As explained above, each task of the system is written as a PALSSware app, instead of a standalone application program on OS as in Fig. 1.1(a). Specifically, each app is written as a C program module that are linkable with PALSSware, which is also a C module, to produce a complete application program on OS.

A simplified code of our PALSSware implementation is given in Fig. 2.3 for a more concrete understanding of its functionalities. As shown in the code, the PALSSware code has the `main` function that contains the main loop, while the app side is supposed to provide the implementation of the `job` function, which is declared in the header `app.h`. Also, `config.h` declares common information of the system, such as

```

1      // palsware.c
2
3      #include "app.h"
4      #include "config.h"
5      ...
6
7      inbox_t *cur_inbox, *nxt_inbox;
8
9      void pals_send(char tid, char *msg) {
10         ...
11     }
12
13     int main() {
14         uint64_t sync_time = PALSware_initialize();
15
16         while(true) {
17             sleep_until(sync_time);
18             fetch_msgs(sync_time);
19             job(cur_inbox);
20             sync_time += PALS_PERIOD;
21             swap_inbox();
22         }
23     }

```

Figure 2.3 Structure of PALSware implementation

the synchronization period and IP addresses for the tasks and multicast groups.

The program starts with an initialization process. It initializes the internal state of the program, as well as opening two network sockets, each for TX and RX. It also sends join messages for multicast groups if necessary. At the end, it returns the first synchronization time for the forthcoming main process. The value of the time represents the number of nanoseconds elapsed after the Unix Epoch time, and every synchronization time is divisible by the period length.

In the main loop, the program sleeps until the given synchronization time. When the time comes, the program wakes up and runs the `fetch_msgs` function, which fetches incoming packets from the RX socket, parses each packet, and stores it in an appropriate inbox. There are two inboxes in the memory at runtime: `cur_inbox` and

`nxt_inbox` for storing messages for the current period and the next period, respectively. As a result, `cur_inbox` contains all messages for the job of the current period, and `nxt_inbox` contains messages for the next period that arrived early.

Now, the `job` function from the app code starts to run with the messages of `cur_inbox`. The `job` function may send a message to other tasks by calling the `pals_send` function of the PALSware code, with a task ID (or a multicast group ID) for the destination and a memory pointer to the message byte array. The destination ID is converted to the corresponding IP address by PALSware, according to the configuration data.

After the execution of `job` is done, the next synchronization time is computed, by adding the period length to the current synchronization time. Finally, the `swap_inbox` function swaps inboxes, so that `nxt_inbox` becomes `cur_inbox`, and then it empties the new `nxt_inbox`. Then the program repeats the main loop for the next period.

Simplifications In this work, we simplified several aspects of the original work of PALSware[7] to ease the verification process. First, we restrict all tasks to have a common synchronization period, while the multi-rate and multi-phase extensions in the original work allows different periods for each task as well as division of a period into several phases. Another restriction is that we allow the system to send at most one message for each sender-receiver pair in a single period. This restriction does not lose too much generality, because a general system can be simulated by a restricted version of the system that defers all message transmissions until the end of the job, and then pack them into big single messages for each destination, as long as the message length bound permits. Nevertheless, we believe that removing those restrictions is feasible for the future work.

```

1   CoInductive itree (E: Type -> Type) (R: Type): Type :=
2   | Ret (r: R)
3   | Tau (t: itree E R)
4   | Vis (A: Type) (e: E A) (k: A -> itree E R).

```

Figure 2.4 Simplified definition of interaction tree in Coq

2.2 Interaction Trees

Interaction trees are a data structure that represents program behaviors that interact with outer environment, in an abstract and executable way. Our framework uses this form to express abstract specifications. The executable property enables testing of these abstract specifications.

We introduce a simplified definition of the interaction tree data structure in Coq as shown in Fig. 2.4. The data type `itree` has two index types `E` and `R`, where `E` represents the type of events, indexed by the return type of each event, and `R` represents the return type of the tree.

Interaction trees are constructed with the three constructors `Ret`, `Tau`, and `Vis`. The first constructor `Ret` represents the base tree which returns `r`. The second one `Tau` constructs a tree that takes a silent step to be the given tree `t`. This `Tau` case is essential for representing recursive trees. Finally, `Vis` constructs a tree that produces a visible event `e` that returns a value of the type `A`, and then proceeds to the continuation `k` that depends on the return value.

One advantage of using interaction trees is that it supports convenient way of combining multiple trees. For example, the interaction tree library provides `bind` operator of the following type:

$$\text{bind} : \text{forall } (A B : \text{Type}), \text{itree } E A \rightarrow (A \rightarrow \text{itree } E B) \rightarrow \text{itree } E B$$

so that together with `Ret` we can write interaction trees in a monadic style. We use

1	Definition ctrl_spec (st: state)	1	Definition ctrl_spec (st: state)
2	: itree sendE state :=	2	: itree sendE state :=
3	let (st1, tid1) = update st in	3	(st1, tid1) <- update st ;;
4	Vis _ (SendEvent tid1 grant_msg)	4	SendEvent tid1 grant_msg ;;
5	(fun _: unit =>	5	send_hb st1 ;;
6	send_hb st1 ;; Ret st1)	6	Ret st1

Figure 2.5 An example interaction tree, the exact definition and a monadic-style representation

the notation $(_ \leftarrow _ \;; _)$ for the bind operator. For example, we can write $r \leftarrow t \;; k \ r$ for bind t k . Also, we may write $t \;; k$ when the return value of t is not used.

In the rest of paper, we express interaction trees in the monadic style, as shown in Fig. 2.5. The example interaction tree is a simplified version of a part of the specification from our first case study. The Coq code in the left side shows the definition of the interaction tree in Coq. The tree is parametrized by the state st . It first updates the state with a mathematical function `update` that returns the next state $st1$ and a task ID $tid1$. Then, the tree produces an event of `SendEvent` of the type $(sendE \ unit)$ that returns the unit value, followed by another interaction tree `send_hb`. Finally, the tree returns the new state $st1$. The right side of the figure shows the monadic-style representation of the interaction tree. This is possible since both pure calculations and event generations can be lifted to interaction trees. We will clarify the exact definitions when a precise distinction between them is necessary.

Also, to hide complexities of dependent types, we fix the type of all events as an (unindexed) type of name `EvtCall`, rather than indexed type such as $E \ A$ above, and use `RetType : EvtCall \rightarrow Type` to get the return type of an event, for presentation. For example, following this convention, `SendEvent` is of type `EvtCall`, and `RetType(SendEvent) = unit`. Details will be presented in the formalization chapter.

Chapter 3

Overview

In this chapter, we give a high-level overview of VeriPALS. We begin with the overall structure of our verification framework, with a brief explanation for each component. Then, we list our key ideas for handling real-time and distributed features in formal modeling and verification.

3.1 Framework

The diagram in Fig. 3.1 shows the overall structure of the VeriPALS framework. First, as shown in the upper-left box, the framework takes application-specific data as parameters and assumptions from the user, which consists of C modules, abstract specifications, and simulation proofs between them. In the figure, these user-given objects are identified with dashed borders.

From this input, the framework constructs the real-world model shown on the lower-left side of the figure. This is a mathematical model for the execution of the real-world system, in which the network and the real-time operating system are con-

servatively modeled to include all possible behaviors of them in the real world. The operating system model runs an application program written in C, which results from linking PALSware with an app.

Above the real-world model is the executable abstract synchronous model, or the abstract model for short, that generates the synchronous behaviors of the given application system. The model is written as an interaction tree, so that it supports both formal verification in Coq and testing of the system. Note that all objects highlighted in gray in the figure are executable specifications written as interaction trees.

Finally, the behavioral refinement between the real-world model and the abstract model are proved in multiple steps, vertically composing the refinement proofs (Ref.1 - Ref.5) involving the intermediate system models shown on the right side of the figure.

In the following paragraphs, we will explain main components of the framework with more details.

Parameters and Assumptions In the box of Parameters & Assumptions, the white boxes denote the PALSware apps written in C, which must comply with the interface of PALSware. The gray boxes denote the abstract specifications for each app. The specifications are written in interaction trees. Finally, the vertical tilde symbols between apps and specifications represent simulation proofs, which imply that every execution of the given app can be simulated by the specification.

Note that manual writing of the specifications for the apps are not mandatory for using the framework. For testing, we support direct linking of the C programs with the generic part of abstract model through the OCaml foreign function interface with C. Also, we provide a special compiler that generates an interaction tree specification from any given app. We plan to make it automatically generate a simulation proof

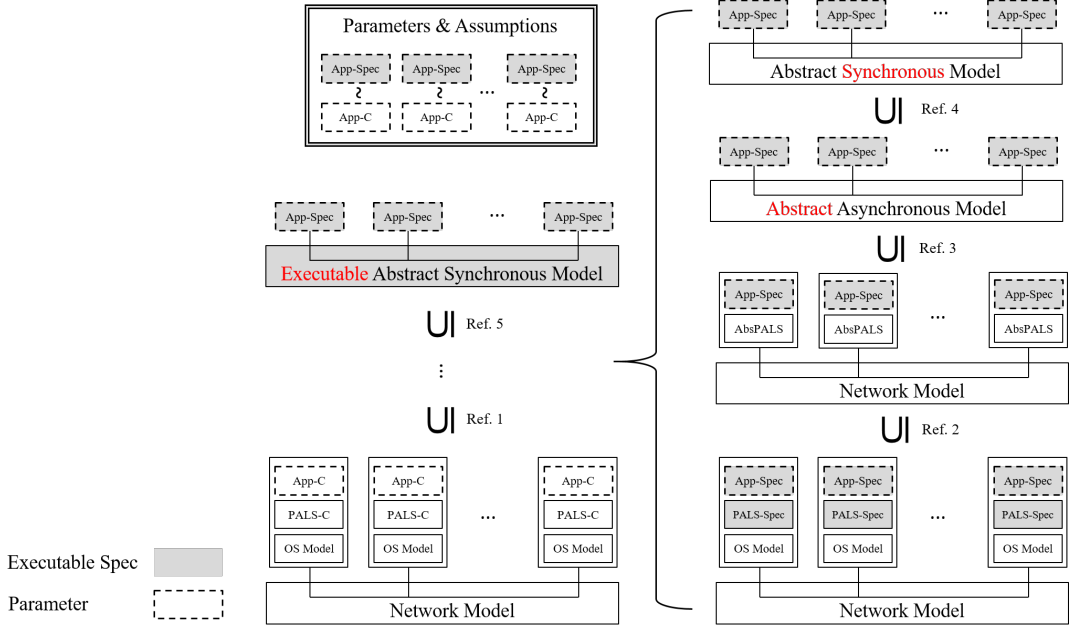


Figure 3.1 The overall structure of the verification framework

between them as well¹.

Network Model The network model is a mathematical description of the behavior of the network in the real-world physical environment. We assume a reliable network with the maximum transmission time μ , and the model is designed to include all possible behaviors under this assumption. A node on the network may send a message packet that contains the sender's IP address, the destination IP address, the port number, and the actual byte-array payload. Then, the message arrives at the destination node within μ .

In addition, the model supports multicasting, *i.e.*, sending a single message to multiple destinations, which is used by PALSware. A node may join to a multicast group by sending a *join* packet to the associated *group IP address*. We also assume

¹We expect this proof generation (*i.e.*, verification of the compiler) to be done within a few weeks.

that the join process is completed within the time μ . The IP addresses from 224.0.0.0 to 239.255.255.255 are dedicated to such multicast group IP addresses, according to the IPv4 protocol.

Operating System Model The real-time operating system model contains several functional modules that work between PALSware and the network. First, it provides a local clock whose skew is bounded by the maximum clock skew ε . Also, the model includes a network socket module for accessing the network. An application program may open a socket and apply further socket operations such as send, receive, and bind. In addition, the model offers a timer service module that provides the sleep functionality, whose wake-up delay is bounded by a constant δ .

For precise modeling, the operating system model permits random failures at any time, as the real-world system also encounters unpredictable failures. The model follows the fail-stop assumption: in the case of failure, the operating system stops running, and may recover nondeterministically in the future.

An application program on OS is modeled as an event-generating transition system. In our case, we first use the Clight formal semantics to specify the behaviors of an arbitrary C program as an application program, and then instantiate the program with the linked program PALSware and one of the PALSware app.

Executable Abstract Synchronous Model The executable abstract synchronous model is expressed as an interaction tree that generates the synchronous behaviors of the application system by performing the following steps. The model consist of a main interaction tree that controls the whole system, and multiple specifications where each describes a distributed component of the application system. At the beginning of each period, the model sequentially executes the tasks one by one with their inbound messages. Note that the execution order does not affect the overall

system behavior, because interactions between the tasks are done only in between the periods. After all the jobs are done, the resulting output messages are gathered and then distributed to their destinations. After that, the model moves to the next period and repeats the aforementioned process.

For formal verification, proving a behavioral property on the synchronous model greatly reduces the complexity over proving it directly on the real-world system model, since components of the real-world model such as the operating system model are too detailed to carry out proofs about a concrete system on it. The behavioral refinement proof between the two models guarantees that a formally proven property on the synchronous model also holds on the real-world system model.

For testing, the framework can generate an executable program of the abstract model. If the specifications for each app is given from the user, then the specifications and the generic abstract model can be linked in Coq and then extracted to OCaml. Otherwise, we can apply one of the two aforementioned methods: using OCaml foreign function interface with C, or using our compiler to interaction tree. The first method runs faster, while the second one has a smaller gap between the formal development and the executable program.

Refinement Proof using Intermediate Models Rather than proving the behavioral refinement between the real-world model and the abstract model at once, we defined multiple intermediate models that gradually abstract the concrete details of the real-world system model. The behavioral refinement proof for each step is denoted as one of the subset symbols named from Ref. 1 to Ref. 5 in Fig. 3.1, where the end-to-end behavioral refinement is deduced by the transitivity of the refinement relations.

The abstraction done in each step is as follows. In Ref. 1, the C program details are abstracted away, such as the concrete C program syntax and memory states at

runtime. In Ref. 2, the operating system model and the specification of PALSware are merged into one entity that we call abstract PALSware, removing detailed operating system functionalities including the local clock. Ref. 3 replaces the concrete network model and the abstract PALSware with an abstract model, where job execution timings are still asynchronous. Then, in Ref. 4, we show that the abstract asynchronous model can be simulated by a synchronous model where all jobs in a period are done in a single step, at the exact time. Finally, by Ref. 5 we reach the top-level model, which enables testing of the application system.

3.2 Key Ideas

In this chapter, we explain our key ideas in building the VeriPALS framework. The main challenge of the work is about how to deal with real-time and distributed features in formal modeling and verification. We summarized the challenge into several topics, and for each topic we describe the problem and our solution.

3.2.1 Concurrent Executions of Nodes

Problem Our real-world model must express concurrent executions of each node connected to the network, as the real-world system does. Especially, the model should specify the exact time of each observable event, in order to reason about the correctness of real-time systems. Interactions between nodes is done by exchanging messages. Each node may send a message to the network in any time, then within some amount of transmission time, the network delivers the message to its destination.

Solution In order to formally model this structure, we first assume an atomic time unit, and the real-world model take a transition step for each tick of the time unit. Specifically, each step is done in the following way. First, the network model chooses message packets to distribute now. Then, each node takes a step, in which incoming

message packets are processed and an observable event may occur, possibly releasing a packet to the network. Finally, the network gathers all the released packets.

As the length of the time unit approaches to zero, the model becomes more precise in expressing the behaviors in the real world. In addition, we assume that the length exactly divides the length of one nanosecond, to ease the proof. In our development, we axiomatized λ , which is the number of the time units in one nanosecond, since the exact value is insignificant in the verification.

3.2.2 Global Clock vs. Local Clock

Problem Each node in the real-world model should maintain a local clock that satisfies the following constraints: (1) the skew against a global clock must be bound within the value ε , and (2) the clock must be monotone: when the value of the local clock changes, it is always increasing. The unit of local clock is the nanosecond, and we assume the value of ε is also given in nanoseconds. The global clock is given in the atomic time unit in the model, as described above.

Solution Our local clock model in a node works in the following way. First, when the operating system is done booting, the initial local clock value is nondeterministically chosen within the clock skew ε from the current global clock. Here, we convert the unit of the global clock from the atomic time unit to nanoseconds by rounding down. Then, in next steps, we again nondeterministically choose the next local clock that meets both of the constraints.

Once we have a valid local clock value that meets the constraints, there always exists a valid value of the next local clock. The informal proof is as follows. If the next global clock in nanoseconds is not increased, we can keep the local clock unchanged. If the global clock in nanoseconds is increased by one, we can also increase the local clock by one. In either way, the amount of skew is unchanged.

3.2.3 Real-time Local Executions of Node Model

Problem The node model should express the real-time cooperative behavior of the operating system and its application program. Specifically, when a program is running on the operating system, it may either take an internal step or invoke an external function call. If the external function is an API function for an OS system call, the operating system takes over and process the request, such as sending a message to the network. Otherwise, the external function call is considered as an observable event, and we assume any well-typed return value can be given. In addition, we also want to take care of node failures that may happen in any time.

Especially, for reasoning about time, state transitions of the node model should involve latency. The time delay is nondeterministic due to various factors, such as cache miss, page fault, context switch, and so on. Importantly, the delay in execution of the application program is not necessarily related to the number of its internal steps. It's because, in addition to the nondeterministic factors listed above, our formal C semantics is designed as an abstract machine that is not relevant to the actual physical machine. For example, multiple steps of C program execution may correspond to nop in the physical machine (*e.g.*, in processing the 'sequence' case of the program syntax), and a single step may correspond to a significant length of machine code (*e.g.*, in evaluating a complex expression).

Solution We model the transition steps of a node as follows. While the operating system is turned on, the state of the node carries a natural number that we call *latency count*, which indicates the number of latency steps left. In a transition step, the node may fail nondeterministically. In that case, we assume the fail-stop behavior: the node silently stops running and may recover randomly in the future. Otherwise, the transition depends on the latency count. If the latency count is positive, the transition step reduces the number, keeping the rest of the state unchanged. When the latency

count reaches zero, the actual internal transition occurs and then the latency count is reset to a random number. Modeling latency in this way reduces nondeterminism in transition steps and also enforces the liveness of each node’s execution. Until the latency count becomes zero, next several steps are deterministic unless it fails in the middle of the steps. After the fixed number of count-reducing steps, the node performs the next transition.

The internal transition of a node works as follows. First, if the operating system has control, it continues the current system call procedure, following the rule of our formal model of the operating system. Our operating system model provides the network and timer services, for the details of which we refer the readers to our Coq development[23]. Second, if the application program has control, it initially takes an arbitrary number of (silent) program-internal steps. By doing this, we remove the dependence between the time delay of latency and the number of program steps. And then, the *effective step* of the program occurs as the following. If the current program state is invoking an external function call, the node checks the external function. If it is an API function for a system call, the node initiates the corresponding OS system call procedure. Otherwise, the function call is regarded as an observable event, and the node provides an arbitrary return value to the program. If the program state not calling an external function, it takes one internal step unless the state is a final state.

3.2.4 Time Constraint on Network Transmission Times

Problem According to the assumptions on the network, the transmission time of a message packet is bounded by the time length μ . In other words, when a message packet is sent from a node, its delivery should be completed with the time μ . Also, in order to cover every possible behavior, the network model should allow all possible transmission times less than μ . We assume that μ is given in nanoseconds.

Solution When a message packet is released from a node, the network model pairs the packet with an arbitrary value less than $\lambda\mu$, which we call *delivery count*, and stores the packet in the state of the network model. The delivery count works as the latency count of the node model. The count is reduced by 1 for each transition step of the system until it reaches zero, and then the network model delivers the packet to the destination node.

3.2.5 Time Constraint on Program Executions

Problem For the correct synchronization of PALSware, we need to impose a time constraint on every single iteration of the main loop. However, imposing a time constraint on a part of program code is nontrivial; at least, the solution for the transmission time is not applicable here. Unlike the former problem, the execution is done in multiple steps according to the **semantics** of the code, so we have to restrict the sum of the delays from each step, not a delay of a single step. Moreover, in our case, the code contains unknown part given from the user, so that the number of steps is also unknown in verifying the generic system built on PALSware. In the worst case, the user could (mistakenly) provide code that enters to an infinite loop and never returns, in which case the code always fails to meet the constraint.

Therefore, we need to devise a new way to properly reflect the time constraint of program execution in the real-world model, so that it effectively rules out invalid programs that fails to meet the constraint, such as a program with an infinite loop.

Solution To resolve the problem, we adopt a virtual event indicating that a violation of an assumption occurs during execution. This idea is inspired by the work of compiler verification[18] that uses a special `oom` event to indicate the out-of-memory situation in the refinement proof.

In our work, we let the node model produce a `Nobehavior` event when a viola-

tion of an assumption happens during execution, and then rule out all executions that contain `Nobehavior` in the refinement proof. First, we separate each iteration according to the `sleep_until` function calls: an iteration starts when the previous `sleep_until` call returns, and the iteration ends when the program calls `sleep_until` again. Then, the model measure the length of time that each iteration takes, and then produce `Nobehavior` when it exceeds the given constraint length α_{loc} .

Specifically, we add a *time limit* value in the node model. When `sleep_until` returns at the time t_{cur} , the model sets the time limit as $t_{\text{cur}} + \alpha_{\text{loc}}$. After the time limit is set, every step of the node model checks whether the local clock does not exceed the time limit value. If the program calls `sleep_until` before that, the time limit value is reset to none. Otherwise, the node produces `Nobehavior` to indicate the violation. As a result, all executions without `Nobehavior` are valid executions that don't include situations that violate the assumption.

3.2.6 Observable Behaviors of a Real-Time Distributed System

Problem For real-time distributed systems, observable behaviors must specify not only the order of the observable events from an execution, but also the times and locations of them. In our work, there is an issue in representing event occurrence times. In the real-world model, every event produced from one node in an execution has a different occurrence time, since a node step may produce at most one event at a time. However, in the abstract model, all events in a period occur at once (with order), at the synchronization time. For the refinement proof, we need to define the observable behaviors of the real-world model and abstract model in a way that two corresponding executions from the respective models can be identified.

Solution We define an observable behavior of a real-time distributed system as a list of local behaviors, where each local behavior is a coinductive list whose elements

are timestamped observable events produced from a single node.

We parametrized the node model with a timestamp generator, which is a function that converts the local clock to the timestamp value. In our real-world model, we instantiate the timestamp generator with a function that computes the synchronization time that begins the period of the given local clock. In this way, an execution of the real-world model generates the identical behavior with the corresponding execution of the abstract model.

Chapter 4

Formalization

In this chapter, we present the formal structure of our framework. We start from general definitions regarding distributed systems and their real-time behaviors. Then, we present the formal form of application systems, which serves as the parameters and assumptions of the framework, which are supposed to be given from the user. Then, we present our two system models: the real-world model and the abstract model. The former one is a state transition system that includes the network and operating system model, and the latter one is an interaction tree that simulates the ideal synchronous behaviors. Finally, we state the final theorem, which proves the refinement between the two models under an arbitrary application system.

4.1 General Definitions

Events We start from the definitions of events used in the models. Let `EvtCall` be the set of event calls. For example, it can be a concrete external call with a function name and argument values, or an invocation of `Nobehavior` introduced in Section 3.2.5. In the development, we use various types of such event calls, but in

this presentation we unify them as **EvtCall** for simplicity. Also, let **RetType** be the function that maps an abstract event call $ec \in \mathbf{EvtCall}$ to the type of its return values. Then the event type $\mathbf{Evt} = \{(ec, r) \mid ec \in \mathbf{EvtCall}, r \in \mathbf{RetType}(ec)\}$ is the set of pairs of an event call and its return value. Additionally, we define the set of timestamped event as $\mathbf{TEvt} = \mathbb{N} \times \mathbf{Evt}$ and the set of timestamped event lists as $\mathbf{TEvts} = \overrightarrow{\mathbf{TEvt}}$. Like this, we mark list types or list objects with overline arrows.

Distributed Systems Here, we define several kinds of types for distributed system behaviors. First, we define concrete behaviors $\mathbf{ConcBeh} = \overrightarrow{\mathbf{stream}(\mathbf{TEvts})}$ where each infinite stream represents the local concrete behavior from each distributed task. Each element of the stream represents the list of events occurred at each step. Then, to relate the behaviors of two different systems, we use (abstract) behaviors $\mathbf{Beh} = \overrightarrow{\mathbf{colist}(\mathbf{TEvt})}$ that abstracts the concrete step information, where we let $\mathbf{colist}(A)$ be the set of coinductively defined lists of A , which permits infinite lists.

Now, we define a distributed system as a tuple $sys = (S, |\cdot|, \cdot \rightarrow \cdot, I) \in \mathbf{DSys}$ where S is the set of states, $|\cdot|$ gives the number of nodes in the state, $\cdot \rightarrow \cdot \subseteq S \times \overrightarrow{\mathbf{TEvts}} \times S$ is the transition steps, and I is the set of initial states. Additionally, we add a validity condition to the transition steps: the length of event lists should be equal to the number of nodes in the starting state, and the number of nodes should be preserved in every step $\cdot \rightarrow \cdot$.

From this, we define the concrete behaviors $\mathbf{ConcBehState}(s)$, and the behaviors $\mathbf{BehState}(s)$ of a system state s as shown below, where the predicate $\mathbf{isAllValid}(\cdot)$ assures that there is no **Nobehavior** events in it. Note that $\mathbf{ConcBehState}(s)$ is defined coinductively as the greatest fixed point. Putting them together, we define the behaviors of the system $\mathbf{BehSys}(sys)$. Note that we use the symbol $++$ to represent the append operations for list-like objects (*e.g.*, list, stream, and colist), and $++_{\text{each}}$ for the element-wise append operations between two lists of list-like objects. Also, \simeq

denotes the trivial equivalence relation between a concrete behavior and an abstract behavior.

$$\begin{aligned}
& cbeh \in \mathbf{ConcBehState}(s) \iff \\
& \quad (1) \quad s \in \mathbf{StuckState} \wedge |cbeh| = |s| \wedge \mathbf{isAllValid}(cbeh) \quad \vee \\
& \quad (2) \quad \exists e, s', cbeh'. \\
& \qquad s \xrightarrow{e} s' \wedge \mathbf{isAllValid}(e) \wedge \\
& \qquad cbeh' \in \mathbf{ConcBehState}(s') \wedge e \mathrel{++}_{\text{each}} cbeh' = cbeh
\end{aligned}$$

$$\begin{aligned}
& beh \in \mathbf{BehState}(s) \iff \\
& \quad \exists cbeh. cbeh \in \mathbf{ConcBehState}(s) \wedge cbeh \simeq beh
\end{aligned}$$

$$\begin{aligned}
& beh \in \mathbf{BehSys}(sys) \iff \\
& \quad (\nexists s. s \in I) \vee (\exists s. s \in I \wedge beh \in \mathbf{BehState}(s))
\end{aligned}$$

Then the refinement between two distributed systems, namely an abstract system sys_a and a concrete system sys_c , is defined as the subset relation of their behaviors:

$$\mathbf{BehSys}(sys_c) \subseteq \mathbf{BehSys}(sys_a)$$

Proof Technique In proving the refinement between two systems $sys_a, sys_c \in \mathbf{DSys}$, we use the following coinductive multi-step simulation relation between the states, with an ordinal-number stuttering index:

$$\begin{aligned}
& s_a \sim_{(o, e_{\text{acc}})} s_c \iff \\
& (\exists e_c, s'_c. s_c \xrightarrow{e_c} s'_c) \wedge \\
& (\forall e_c, s'_c, e'_{\text{acc}}. s_c \xrightarrow{e_c} s'_c \wedge \text{isAllValid}(e_c) \wedge e'_{\text{acc}} ++_{\text{each}} e_c = e_{\text{acc}} \implies \\
& \quad \exists o'. \\
& \quad (1) \quad o' < o \wedge s_a \sim_{(o', e'_{\text{acc}})} s'_c \quad \vee \\
& \quad (2) \quad \exists e_a, s'_a. s_a \xrightarrow{e_a}^+ s'_a \wedge e_a = e_c \wedge s'_a \sim_{(o', \text{repeat}(\square, |s'_c|))} s'_c
\end{aligned}$$

where $\text{repeat}(a, n)$ denotes a list produced by repeating a for n times. We use an ordinal number for the stuttering index in order to cover the cases when the number of stuttering steps is unknown upfront.

Intuitively, the simulation relation implies that, for any execution from s_c , it eventually reaches to another state s'_c , and the accumulated events e_{acc} can be simulated by a multiple step $s_a \xrightarrow{e_{\text{acc}}}^+ s'_a$. The case of (1) accumulates the events, and the case of (2) specifies the multiple step of s_a .

We proved that the simulation relation implies the refinement of two states.

Lemma 4.1.1 (Adequacy of Simulation). *For any two states $s_a \in S_a$ and $s_c \in S_c$ and an ordinal number o , if the simulation holds with empty accumulated traces, i. e., $(s_a \sim_{(o, \text{repeat}(\square, |s_c|))} s_c)$, then the refinement $\text{BehState}(s_c) \subseteq \text{BehState}(s_a)$ holds.*

4.2 Application System of the Framework

At first, we declare several sets regarding the Clight formal semantics: the set of C programs CProg , the set of C functions CFunction , the set of memories Memory , and the set of C program states CProgState at runtime. The formal definitions of them are in the Coq development of CompCert.

Also, we denote the set of interaction trees as $\text{ITree}_{(E, R)}$ where E is the set of possible event calls from the trees and R is the set of return values. In this presentation, we fix $E = \text{EvtCall}$ so we omit it from now on, while R can be changed depending on

the situation. Additionally, we define a set of message box $\text{MsgBox} = \overrightarrow{(\text{Bytes})^?}$ that contains a list of optional byte-list messages.

Now, we define the form of abstract specifications that appears in the parameters of our framework. An abstract specification for each app is given as an element of the following set:

$$\begin{aligned} \text{spec}_{\text{app}} \in \text{AppSpec} &= \{(S_{\text{app}}, \text{itr}_{\text{job}}, s_{\text{app_init}}) \mid S_{\text{app}} \in \text{Type}, \\ &\quad \text{itr}_{\text{job}} \in \text{MsgBox} \rightarrow S_{\text{app}} \rightarrow \text{ITree}_{S_{\text{app}}}, s_{\text{app_init}} \in S_{\text{app}}\} \end{aligned}$$

where S_{app} is the type of abstract states of the given task, and itr_{job} is the interaction tree of the task parametrized by the inbox and the abstract state at the beginning of the job. When an interaction tree finishes the execution, it is in the form of $\text{Ret}(s_{\text{app}})$ for a final state $s_{\text{app}} \in S_{\text{app}}$. Finally, $s_{\text{app_init}}$ denotes the initial state.

Simulation Relation First, we define a standard form of simulation relation between an interaction tree $\text{itr} \in \text{ITree}_{\text{unit}}$ and a C program state $\text{cst} \in \text{CProgState}$ with a natural-number stuttering index i : $\text{itr} \sim_{\text{ITr}(i)} \text{cst}$ (for the details of which, refer to the Coq development). The simulation relation is easily lifted from CProgState to CProg such as $\text{itr} \sim_{\text{ITr}(i)} \text{cprog}$.

From this, we define a simulation relation between a PALSware app and its abstract specification as below, which is the user's proof obligation:

$$\begin{aligned}
& spec_{app} \sim_{app} cprog_{app} = \\
& \exists f_{job}. ("job", f_{job}) \in \mathbf{functions}(cprog_{app}) \wedge \\
& \forall (k_{itr} : S_{app} \rightarrow \mathbf{ITree}_{unit}), (i : \mathbb{N}), (k_c : \mathbf{CCont}), \\
& (ptr_{inb} : \mathbf{MemPtrVal}), (m : \mathbf{Memory}), (inb : \mathbf{MsgBox}), (s : S_{app}). \\
& m \in \mathbf{MWRegionValid} \wedge inb \sim_{inbox} (m, ptr_{inb}) \wedge s \sim_{app_state} m \wedge \\
& (\forall s', m'. \\
& \quad (m, m') \in \mathbf{MWRegionUnchanged} \wedge s' \sim_{app} m' \implies \\
& \quad k_{itr}(s') \sim_{ITr(i)} \mathbf{ReturnState}(\mathbf{Vundef}, k_c, m')) \implies \\
& \exists (i_{job} : \mathbb{N}), \\
& \quad (s' \leftarrow itr_{job}(inb, s); ; k_{itr}(s')) \sim_{ITr(i+i_{job})} \mathbf{CallState}(f_{job}, [ptr_{inb}], k_c, m)
\end{aligned}$$

We explain the construction of the relation \sim_{app} as follows. First, \sim_{app} guarantees that a C function of the name “job” and the body f_{job} exists in the C implementation of app. Also, for any inbox inb and abstract state s , if the memory m stores (1) a valid data for the middleware region to which only the PALSware code have access (indicated by $\mathbf{MWRegionValid}$), (2) matched data for inb in the region pointed by a pointer ptr_{inb} (indicated by $inb \sim_{inbox} (m, ptr_{inb})$), and (3) matched data for s in the app region (indicated by $s \sim_{app_state} m$), then the job interaction tree invoked with inb and s are in the simulation relation $\sim_{ITr(i+i_{job})}$ with the call state of C that invokes f_{job} with m and ptr_{inb} , under the assumption that matched continuation after the job in both the interaction tree and C state are in the simulation relation. Here, $\mathbf{MWRegionUnchanged}$ indicates that the middleware region is unchanged.

An application system of the framework is represented as a triple $(prd, \overrightarrow{mc}, \overrightarrow{task}) \in \mathbf{AppSys}$, where $prd \in \mathbb{N}$ is the synchronization period of the system, $mc \in \mathbf{IP} \times \overrightarrow{\mathbb{N}}$ is the multicast group information that consists of the multicast group IP address and the members’ task IDs, and $task$ of the type

$$\mathbf{Task} = \{(ip, spec_{app}, cprog_{app}) \mid ip \in \mathbf{IP}, spec_{app} \sim_{app} cprog_{app}\}$$

consists of its IP address, a specification and an implementation of the task, and the simulation relation between them.

4.3 Real-World Model

From the given application system, the framework constructs the corresponding real-world model. The model includes the network model and the operating system model as its components. The model depends on the system parameters μ , ε , and δ that we explained in the Section 2.1.2.

4.3.1 Network Model

We start from the formal definition of network packets. There are two kinds of packets: message packets and multicast group join packets. The set of message packets are defined as $\text{MsgPkt} = \text{IP} \times \text{IP} \times \text{Port} \times \text{Bytes}$, which contains the sender IP, destination IP, destination port, and the actual payload. On the other hand, a join packet is modeled as a pair of a multicast IP of a group and a local IP address that wants to join the group. As a result, the set of packets is defined as a disjoint union $\text{MsgPkt} \uplus \text{JoinPkt}$.

In this model, a network state is a triple (G, M, J) where $G \in \overrightarrow{\text{IP} \times \text{IP}}$, $M \in \overrightarrow{(\text{IP} \times \text{MsgPkt}) \times \mathbb{N}}$, and $J \in \overrightarrow{(\text{IP} \times \text{IP}) \times \mathbb{N}}$ are as follows:

- G is the table of multicast member information, where each entry $(ip_{\text{mcast}}, ip_{\text{mem}})$ is a pair of a multicast group IP and a member IP. Note that G can also be seen as the list of join packets that have been processed by the network.
- M is the list of message packets in the network, where each entry $((ip_a, m), d) \in M$ consists of the actual destination of message ip_a , the message packet m , and the remaining delivery count d . Here, the actual destination IP differs with the original destination IP in m when the original destination is a multicast group

IP, in which case the message packet is copied for each members of the group. Finally, d is the number of remaining steps until the delivery is completed. When the message packet is initially introduced to the network, the delivery count is set to a random value less than $\lambda\mu$, and then each of the following steps reduces the value by 1.

- J is the list of join packets in the network. Each entry $((ip_{\text{mcast}}, ip_{\text{mem}}), d) \in J$ is also tagged with a delivery count d that works in the same way as M . When the count becomes zero, the entry moves from J to G .

In Fig. 4.1, we present the formal rules for the network steps described in Section 3.2.1. The first step that chooses packets to distribute is expressed as a computable function `to_distrib`, in which the delivery count is reduced by one and packets with zero counts are processed immediately. The second step that gathers output packets starts with classifying the packets into message packets and join packets. Each message packet is copied for each of the actual destination IP address, by `attach_actual_dest`. Then, each packet is tagged with a random delivery count that is valid according to μ . Finally the newly introduced message packets and join packets (tagged with actual destination addresses and delivery counts) is appended to the network state.

4.3.2 Generic System Model on Network

Building on the network model, we construct a formal distributed system from a list of generic nodes. A (generic) node is a tuple $(ip, S, I, step)$ where $ip \in \text{IP}$ is the IP address, S is the set of node states, $I \subseteq S$ is the set of initial node states, and $step \in \mathbb{N} \times S \times \overrightarrow{\text{MsgPkt}} \times (\mathbb{N} \times \text{Evt})? \times S \times (\text{Pkt})?$ is the transition steps. In later descriptions, we also write $(t_{\text{glob}}, s, d, e, s', p) \in step$ as $t_{\text{glob}} \vdash (s, d) \xrightarrow{e} (s', p)$ for readability.

$$\begin{array}{l}
\text{to_distrib}(G, M, J) = \\
\quad \text{let } (M', D) = \text{reduce_dcnt}(M) \text{ in} \\
\quad \text{let } (J', G') = \text{reduce_dcnt}(J) \text{ in} \\
\quad ((G ++ G', M', J'), D) \\
\\
\text{classify_packets}(P) = (P_{\text{msg}}, P_{\text{join}}) \\
\text{attach_actual_dest}(G, P_{\text{msg}}) = P'_{\text{msg}} \\
\quad (P'_{\text{msg}}, M_{\text{new}}) \in \text{ValidDCnts}(\mu) \\
\quad (P_{\text{join}}, J_{\text{new}}) \in \text{ValidDCnts}(\mu) \\
\hline
(G, M, J) \rightarrow_{\text{Gather}(P)} (G, M ++ M_{\text{new}}, J ++ J_{\text{new}})
\end{array}$$

Figure 4.1 Formal semantics of the network model

Then, for a given nodes \vec{n} , we define a state of the generic model as a triple $(t_{\text{glob}}, N, \Sigma)$ where $t_{\text{glob}} \in \mathbb{N}$ is the current global clock time in the atomic time unit, N is a network state, and Σ is the list of node states for each of \vec{n} . The formal transition step of the generic model is shown in Fig. 4.2, in which $D|_{ip}$ represents the messages of D filtered by the destination IP ip .

$$\begin{array}{c}
\text{to_distrib}(N) = (N_1, D) \\
\forall i < |\Sigma|. \quad t_{\text{glob}} \vdash (\Sigma[i], D|_{\text{ip}(\Sigma[i])}) \xrightarrow{E[i]} (\Sigma'[i], P[i]) \\
\quad N_1 \rightarrow_{\text{Gather}(P)} N' \\
\hline
(t_{\text{glob}}, N, \Sigma) \xrightarrow{E} (t_{\text{glob}} + 1, N', \Sigma')
\end{array}$$

Figure 4.2 Formal semantics of generic systems on network

From the definitions of states and steps above, we define a construction of the formal distributed system $\text{SysOfNodes} : \mathbb{N} \times \overrightarrow{\text{Node}} \rightarrow \text{DSys}$ from an initial global clock time and list of nodes to a distributed system. For the other components of the distributed system that are unspecified here, we refer the reader to our Coq development.

4.3.3 Operating System Model

In the real-world model, we instantiate the nodes of the generic model with our operating system model.

Application Program Model The operating system model is parametrized by the application program, which is a transition system $(S, I, F, - \rightarrow -, - \uparrow_{(\cdot)}, - \downarrow_{(\cdot)}) \in \mathbf{Prog}$ where S is the set of program states, $I, F \subseteq S$ are the initial and final program states respectively, $- \rightarrow - \subseteq S \times S$ is the internal steps of the program, $- \uparrow_{(\cdot)} \subseteq (S \times \mathbf{EvtCall})$ is the predicate for program states of event calls, and $- \downarrow_{(\cdot)} \subseteq (S \times \mathbf{Evt} \times S)$ is the program step after event calls. In order to instantiate the program with a C program, we define a conversion $\mathbf{ProgOfCProg} \in \mathbf{CProg} \rightarrow \mathbf{Prog}$ based on the Clight formal semantics of CompCert, where we omit the details here.

Process Model From the program model above, we define the *process* model, which represents the program's runtime state with the state of associated OS resources.

Given a program, we define a normal process state as a pair $(s_{\text{os}}, s_{\text{prog}})$, where s_{os} is the associated OS state and $s_{\text{prog}} \in S_p$ is the current program state. s_{os} is a tuple $(skts, tmr, tlim, sts)$ where $skts \in \overrightarrow{\mathbf{Socket}}$ is a list of sockets being used by the program, $tmr \in \mathbf{Timer}$ is a timer state that the program manipulates, $sts \in \{\mathbf{Idle}\} \uplus \{\mathbf{Processing}(ec) \mid ec \in \mathbf{EvtCall}\} \uplus \{\mathbf{Return}(e) \mid e \in \mathbf{Evt}\} \uplus \{\mathbf{Waiting}(t) \mid t \in \mathbb{N}\}$ is the current status of the operating system for the process. Then, we let a process state is either a normal process state or \perp , which represents erroneous cases (*i.e.*, the program state gets stuck).

The formal transition steps of process is depicted in Fig. 4.3. Under an environment that provides the local clock, the process step changes the process state, during which an observable event and an output packet can be generated. Each step is *effective* so that it enforces progress of the state. In the step rules, we use a boolean-valued

$$\begin{array}{c}
\text{(PROGRAMSTEP)} \\
\frac{\text{OS_idle}(s_{\text{os}}) = T \quad s_{\text{prog}} \rightarrow^+ s'_{\text{prog}}}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{\cdot} \text{Proc} ((s_{\text{os}}, s'_{\text{prog}}), \cdot)} \\
\\
\text{(FINAL)} \\
\frac{\text{OS_idle}(s_{\text{os}}) = T \quad s_{\text{prog}} \rightarrow^* s'_{\text{prog}} \quad s'_{\text{prog}} \in \text{ProgFinal}}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{\cdot} \text{Proc} ((s_{\text{os}}, s'_{\text{prog}}), \cdot)} \\
\\
\text{(OSCALL)} \\
\frac{\text{OS_idle}(s_{\text{os}}) = T \quad s_{\text{prog}} \rightarrow^* s'_{\text{prog}} \quad s'_{\text{prog}} \uparrow_{ec_{\text{os}}} \quad s_{\text{os}} \uparrow_{ec_{\text{os}}} s'_{\text{os}}}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{\cdot} \text{Proc} ((s'_{\text{os}}, s'_{\text{prog}}), \cdot)} \\
\\
\text{(OSRETURN)} \\
\frac{s_{\text{os}} \downarrow_{(ec_{\text{os}}, v)} s'_{\text{os}} \quad s_{\text{prog}} \downarrow_{(ec_{\text{os}}, v)} s'_{\text{prog}}}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{\cdot} \text{Proc} ((s'_{\text{os}}, s'_{\text{prog}}), \cdot)} \\
\\
\text{(EVENT)} \\
\frac{\text{OS_idle}(s_{\text{os}}) = T \quad s_{\text{prog}} \rightarrow^* s^1_{\text{prog}} \quad s^1_{\text{prog}} \uparrow_{ec_{\text{sys}}} \quad s^1_{\text{prog}} \downarrow_{(ec_{\text{sys}}, v)} s'_{\text{prog}}}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{e} \text{Proc} ((s_{\text{os}}, s'_{\text{prog}}), \cdot)} \\
\\
\text{(STUCK)} \\
\frac{\text{OS_idle}(s_{\text{os}}) = T \quad s_{\text{prog}} \rightarrow^* s'_{\text{prog}} \quad s'_{\text{prog}} \in \text{StuckState}}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{\cdot} \text{Proc} (\perp, \cdot)} \\
\\
\text{(OSSSTEP)} \\
\frac{s_{\text{os}} \rightarrow_{\text{os}} (s'_{\text{os}}, p)}{t_{\text{loc}} \vdash (s_{\text{os}}, s_{\text{prog}}) \xrightarrow{\cdot} \text{Proc} ((s'_{\text{os}}, s_{\text{prog}}), p)} \\
\\
\text{(ERROR)} \\
\frac{}{t_{\text{loc}} \vdash \perp \xrightarrow{e} \text{Proc} (\perp, p)}
\end{array}$$

Figure 4.3 Transition steps of process

function `OS_idle` that checks whether the current OS status `sts` is `Idle`.

We first explain the upper four step rules, which are unrelated to the OS resources. The first `PROGRAMSTEP` case describes the transition rule when the application program has control and only program-internal steps are taken. The `EVENT` case is applicable when the program calls an external function of the application system's observable event. Then, the model returns arbitrary value for the event call to the program. The `FINAL` step is taken when the program state can silently reach to a final state. The `STUCK` case can be taken when the program state reach to a stuck state, in which case the process changes from a normal state to \perp .

The next three steps are about manipulating OS resources. The OSCALL step occurs when the program calls an external function that is an OS interface. Then, the OS status sts changes from **Idle** to **Processing**(ec_{os}) in s'_{os} . The OSSTEP step is taken when the current OS state can take its own step, which implies that the current OS status is not **Idle**. The actual manipulation of the OS resources takes place in this step, such as socket operations or setting the timer. Also, the time limit may be set to a certain value when waking up from a sleep. Note that the OS step may produce an output packet. For more details of OS steps, refer to our Coq development. The OSRETURN step is taken when the operating system has completed the work and is ready to return the resulting value to the program. In that case, the OS status sts is a form of **Return**(e), and after the return it changes to **Idle**.

Finally, in the ERROR case, the process may produce arbitrary output event and packet.

Operating System Node Model Now we instantiate a generic node model with our operating system model. We let an OS state be either live or turned off. A live state of OS is a triple $(t_{loc}, lat, proc)$, where $t_{loc} \in \mathbb{N}$ is the local clock time (in nanoseconds), $lat \in \mathbb{N}$ is the remaining latency count for taking the next step, and $proc$ is the current process state that we explained above. When turned off, the OS state is just the **None** (\cdot) value.

The formal transition semantics as a node model is given in Fig. 4.4. In the BOOTING case, a new live OS state is produced with a valid local clock t'_{loc} , an arbitrary latency count l , and an initial process $proc$ which contains an initial OS resource state and an initial program state. The LATENCY occurs when the time limit is not violated and the latency count is positive. In that case, the process accepts messages distributed to this node, and advance the local clock under the constraint of maximum clock skew ε . Finally, the latency count is reduced by one. The NODEFAILURE

$$\begin{array}{c}
\text{(BOOTING)} \\
\frac{(t_{\text{glob}} + 1, t'_{\text{loc}}) \in \text{ValidSkew}(\varepsilon) \quad \text{proc} \in \text{InitialProcess}}{t_{\text{glob}} \vdash (\cdot, d) \dot{\rightarrow} ((t'_{\text{loc}}, l, \text{proc}), \cdot)}
\end{array}
\quad
\begin{array}{c}
\text{(LATENCY)} \\
\frac{\text{is_time_limit_over}(t_{\text{loc}}, \text{proc}) = F \quad \text{accept_msgs}(\text{proc}, d) = \text{proc}^1 \quad (t_{\text{loc}}, t'_{\text{loc}}) \in \text{AdvanceLocalClock}(t_{\text{glob}}, \varepsilon)}{t_{\text{glob}} \vdash ((t_{\text{loc}}, l + 1, \text{proc}), d) \dot{\rightarrow} ((t'_{\text{loc}}, l, \text{proc}'), \cdot)}
\end{array}$$

$$\begin{array}{c}
\text{(NODEFAILURE)} \\
\frac{}{t_{\text{glob}} \vdash (\sigma, d) \dot{\rightarrow} (\cdot, \cdot)}
\end{array}
\quad
\begin{array}{c}
\text{(TIMELIMITOVER)} \\
\frac{\text{is_time_limit_over}(t_{\text{loc}}, \text{proc}) = T \quad \tau = \text{get_timestamp}(t_{\text{loc}})}{t_{\text{glob}} \vdash ((t_{\text{loc}}, l, \text{proc}), d) \xrightarrow{(\tau, \text{NB})} ((t_{\text{loc}}, l, \text{proc}'), \cdot)}
\end{array}$$

$$\begin{array}{c}
\text{(RUNPROCESS)} \\
\frac{\text{is_time_limit_over}(t_{\text{loc}}, \text{proc}) = F \quad \text{accept_msgs}(\text{proc}, d) = \text{proc}^1 \quad t_{\text{loc}} \vdash (\text{proc}^1) \xrightarrow{e}_{\text{Proc}} (\text{proc}', p) \quad \tau = \text{get_timestamp}(t_{\text{loc}}) \quad (t_{\text{loc}}, t'_{\text{loc}}) \in \text{AdvanceLocalClock}(t_{\text{glob}}, \varepsilon)}{t_{\text{glob}} \vdash ((t_{\text{loc}}, 0, \text{proc}), d) \xrightarrow{(\tau, e)} ((t'_{\text{loc}}, l', \text{proc}'), p)}
\end{array}$$

Figure 4.4 Formal semantics of the operating system model

case may happen at any time. Then, the OS is turned off, becoming the None (\cdot) state. The TIMELIMITOVER step can be taken when the local clock t_{loc} exceeds the time limit in the OS state. In this case, the node produces a **Nobehavior** event, expressed as NB in the rule. Finally, the RUNPROCESS step is the case when the actual process step is taken. For this step, the latency count must be zero and the time limit is not over. The OS first accepts the arrived messages. Then, the process step is taken, which possibly produces an observable event and an outgoing packet. For the (optional) observable event, the corresponding timestamp value to the current local clock is attached. Finally, the local clock is advanced to a valid value, and the next latency count is randomly set.

From this transition step rules, we can easily construct a node model from a given

IP address $\text{OSNodeOfProg} : \text{IP} \times \text{Prog} \rightarrow \text{Node}$.

Putting them all together, we can define a function

$$\begin{aligned} \text{RealWorldSystem}(app_sys : \text{AppSys}, t_{\text{glob}}^i : \mathbb{N}) : \text{DSys} = \\ \text{SysOfNodes}(t_{\text{glob}}^i, \text{map}(\text{OSNodeOfProg})(app_sys.3)) \end{aligned}$$

that constructs the real-world model from the given application system and initial global clock (represented in nanoseconds for simplicity). Here, we use the notation $(_) . 3$ to indicate the third element of the given tuple, and an implicit coercion from Task to $\text{IP} \times \text{Prog}$ that uses ProgOfCProg .

4.4 Executable Abstract Synchronous Model

The top-level system model of the framework is the executable abstract synchronous model, expressed as an interaction tree. This tree is to be linked with an application system. Then, the `RunAppEvent` event call triggers execution of the apps.

```

1  Fixpoint run_each (task_id: nat) (inbs: list MsgBox)
2    : itree EvtCall (list MsgBox) :=
3    match inbs with
4    | inb :: inbs' =>
5      outb <- RunAppEvent task_id inb ;;
6      outbs <- run_each (task_id + 1) inbs' ;;
7      Ret (outb :: outbs)
8    | [] => Ret []
9
10 CoFixpoint synch_itree_loop (time: nat) (inbs: list MsgBox)
11   : itree EvtCall unit :=
12   set_time time ;;
13   outbs <- run_each 0 inbs ;;
14   inbs' <- distrib_msgs outbs ;;
15   synch_itree_loop (time + T) inbs'
16
17 Definition synch_itree (time: nat)
18   : itree EvtCall unit :=
19   synch_itree_loop time (initial_msgboxes)

```

Figure 4.5 Executable abstract synchronous model

The procedure of `run_each` is in charge of executing each job in the current task. It takes the current task ID to execute, and the list of inboxes for each task as parameters. First, it checks whether the inboxes are nonempty. If it is, then the head element is the inbox for the current task to execute. Then, it generates `RunAppEvent`, which executes the job of `task_id` with the inbox `inb` and returns an outbox `outb`. Then, it recursively runs `run_each` for the remaining tasks until all the inboxes are processed, and finally returns the whole list of outboxes `outb :: outbs`.

From the definition above, we define `synch_itree_loop` as a (coinductively) recursive procedure that corresponds to the periodic repetition of the whole system. It first sets the current time as the given time, which sets the timestamps of observable events to be produced in this period. Then, it runs `run_each` with the given inboxes, from the task ID 0. After that, it rearrange the messages in resulting outboxes `outbs` to compute the inboxes for the next period `inbs'`, by the function `distrib_msgs`. Finally, the procedure is repeated, with the increased time by the synchronization period `T` and the new inboxes `inbs'`.

The top-level interaction tree procedure `synch_itree` just runs `synch_itree_loop` with the initial time and the initial (empty) inboxes. Using this interaction tree, we can define a function

$$\text{SynchSystem} : \text{AppSys} \times \mathbb{N} \rightarrow \text{DSys}$$

that constructs the abstract model from the given application system and initial global clock.

4.5 Result

Finally, we state the final theorem of the framework that proves the refinement between the real-world model and abstract model of the given application system as below, where details of the proof will be discussed in the next chapter.

Theorem 4.5.1 (End-To-End Refinement). *For any application system $app_sys \in \text{AppSys}$ and a global clock $t_{\text{glob}}^i \in \mathbb{N}$, the refinement between the following two systems holds:*

$$\text{BehSys}(\text{RealWorldSystem}(app_sys, t_{\text{glob}}^i)) \subseteq \text{BehSys}(\text{SynchSystem}(app_sys, t_{\text{glob}}^i))$$

.

Chapter 5

Refinement Proof using Intermediate Models

In this chapter, we present a sketch of the refinement proof from the real-world model to the abstract model. The proof is constructed by vertically composing five refinement subproofs, involving four intermediate models. We devote each section of this chapter to each subproof. In each section, when we refer to the two involved system models, we call each the concrete system and the abstract system, respectively.

5.1 Refinement 1: Abstraction of C programs

For the first step, we abstract the real-world model by replacing the C programs of the system with their interaction-tree specifications, on the identical network and operating system model. As a result, complicated details of formal C semantics is removed, such as the concrete C syntax or memory model.

Specifically, the refinement proof in this stage is again constructed by vertically composing multiple refinement subproofs, in each of which a single node is replaced from a C-program node to an interaction-tree node. In order to do that, we first define

a simulation relation between two nodes \sim_{Node} that satisfies the following adequacy lemma:

Lemma 5.1.1 (Adequacy of Node Simulation). *For any two list of context nodes ns_1 and ns_2 and two nodes in the simulation relation $n_a \sim_{\text{Node}} n_c$, the refinement between the following two system holds for any initial global clock t_{glob}^i :*

$$\begin{aligned} & \text{BehSys}(t_{\text{glob}}^i, \text{SysOfNodes}(ns_1 ++ [n_c] ++ ns_2)) \subseteq \\ & \text{BehSys}(t_{\text{glob}}^i, \text{SysOfNodes}(ns_1 ++ [n_a] ++ ns_2)) \end{aligned}$$

.

Then, we define a conversion function from an interaction tree to an OS application program $\text{ProgOfITree} \in \text{ITree}_{\text{unit}} \rightarrow \text{Prog}$ whose detail can be found in the Coq development. The function satisfies the following property:

Lemma 5.1.2 (Program Simulation to Node Simulation). *For any C program $cprog$ and an interaction tree itr , if there exists a natural-number index i such that $itr \sim_{\text{ITr}(i)} cprog$ holds, then the nodes constructed from each of them with an IP address $ip \in \text{IP}$ satisfy the simulation relation:*

$$\text{OSNodeOfProg}(ip, \text{ProgOfITree}(itr)) \sim_{\text{Node}} \text{OSNodeOfProg}(ip, \text{ProgOfCProg}(cprog))$$

.

For the last step, we actually prove that the C program and the interaction tree are in the simulation relation. Let $cprog_{\text{PALS}}$ be the PALSware implementation in C and $itr_{\text{PALS}} : \text{AppSpec} \rightarrow \text{ITree}_{\text{unit}}$ be the interaction tree specification that we wrote for the implementation. Then the following lemma completes the refinement proof, from the application system given from the user. In the lemma, the \oplus symbol is a syntactic linking operator for C programs defined in the CompCert.

Lemma 5.1.3 (App Simulation to Program Simulation). *For any $spec_{\text{app}} \sim_{\text{app}} cprog_{\text{app}}$, there exists i such that $itr_{\text{PALS}}(spec_{\text{app}}) \sim_{\text{ITr}(i)} cprog_{\text{PALS}} \oplus cprog_{\text{app}}$ holds, given that the linkings succeed.*

We give the idea of the simulation proof in Fig. 5.1. In advance of the simulation proof, we define four match relations, denoted as the initial, loop, job_begin, job_end

relations in the figure, between an interaction tree and a C program state. The proof starts from the initial match, and we proceed by checking the simulation until the states are in the loop match case, which indicates that the both states are at the beginning of the main loop. Then, we proceed until the states are in the job_begin match case, where they call the job of the task. Now, the user's obligation $spec_{app} \sim_{app} cprog_{app}$ guarantees that the simulation holds until the job is completed, and when the job is completed, the states are in the job_end match case. The objects marked with dashed line in Fig. 5.1 indicate this part. It also implies that, if the job is never completed (*e.g.*, infinite loop), the simulation holds forever. After returning from the job, we keep proceeding the simulation proof until the states go back to the loop match case. Then, by coinductively we obtain the proof that the simulation relation holds in the entire execution. For more technical details for coinductive simulation proofs, refer to the Paco library[19].

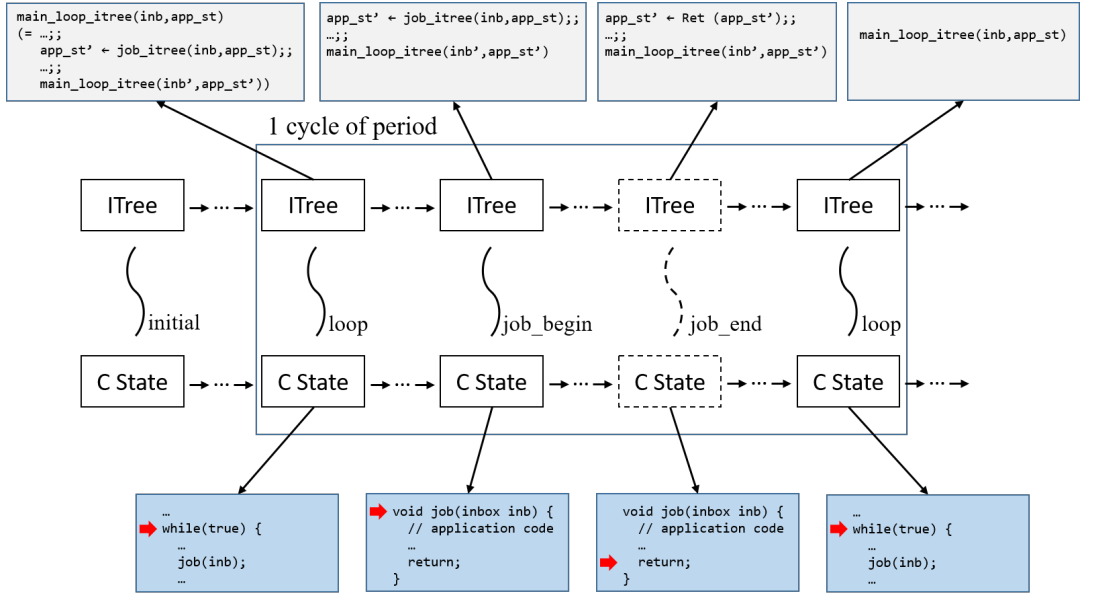


Figure 5.1 Program simulation proof in Ref. 1

5.2 Refinement 2: Abstract PALSware

In this stage, we merge the operating system model and the abstract specification of PALSware itr_{PALS} into a single object called an abstract PALSware node. By doing this, each node is specialized in the behavior of PALSware, removing general-purpose concrete features of the operating system model.

The formal definition is as follows. Under the given application specification $spec_{\text{app}} = (S_{\text{app}}, itr_{\text{job}}, s_{\text{app_init}})$, we define the set of node states as $\{\cdot\} \uplus \text{Prep} \uplus \text{On}$, where \cdot represents the case when the node is off, **Prep** represents the preparation stage of PALSware before going into the periodic executions, and **On** states actually performs the periodic exeuctions of the task.

Specifically, a **Prep** state is a pair $(mids, ps)$ where $mids \in \vec{\mathbb{N}}$ is the multicast IDs to send join packets, and ps is the message packets being accumulated since the node is turned on.

Also, an **On** state is a tuple $(t_{\text{synch}}, ps, inb, sh, itr)$ where $t_{\text{synch}} \in \mathbb{N}$ is the current synchronization time, $ps \in \vec{\text{MsgPkt}}$ is the accumulated message packets, $inb \in \text{MsgBox}$ is the inbox that stores messages that arrives early (due to the clock skew), $sh \in \vec{\mathbb{B}}$ is the send history of the current period, and $itr \in \text{ITree}_{S_{\text{app}}}$ is the current state of the interaction tree. Here, the send history sh is required for limiting the number of messages sent to each task, as stated in the limitation of our current work.

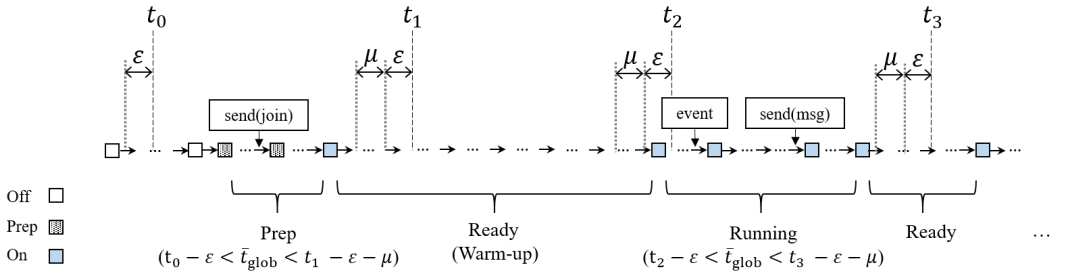


Figure 5.2 AbsPALS

Fig. 5.2 shows how an abstract PALSware node works. Initially the node is turned off. At a random moment, the node is turned on within the range of time $(t_0 - \varepsilon, t_1 - \varepsilon - \mu)$ where t_0 and t_1 are two consecutive synchronization times. In the preparation stage, the node sends multicast join packets until $t_1 - \varepsilon - \mu$, and also receives message packets sent to this node. After sending all the join packets, the node changes to the **On** state and undergoes the warm-up stage: it waits until the time $t_2 = t_1 + T$ to receive all the messages sent to the job at t_2 . In this stage, the interaction tree *itr* in the state is in the form of **Ret**($s_{\text{app_init}}$). After that, within the time range $(t_2 - \varepsilon, t_3 - \varepsilon - \mu)$ the node executes the job *itr*_{job} of the period. The job may start at most ε earlier than t_2 , which makes it possible to simulate the execution of the OS node that involves the clock skew. Also, even though the job is completed before $t_3 - \varepsilon - \mu$, it is enough to simulate the OS node since any execution of the OS node that does not complete the job before this time will generate a NoBehavior event. In the whole process, the node may randomly fail, in order to simulate random failures of the operating system node.

In the refinement proof, we reused the Theorem 5.1.1 again and showed that the node simulation between a **OSNode** node and the corresponding abstract PALSware node holds.

5.3 Refinement 3: Abstraction of Network

In this stage, we simplify the message communication via network to the direct communication. The concrete system of this stage consists of abstract PALSware nodes and the network model, while the abstract system, the abstract asynchronous model, consists of only abstract asynchronous nodes because now transmitted messages are instantly delivered and stored in the destination nodes.

The system state is $(t_{\text{glob}}, \Sigma)$ where $t_{\text{glob}} \in \mathbb{N}$ is the global clock and $\Sigma \in \overrightarrow{\mathbf{AANodeState}}$ is the list of node states of this system. Under an application specification $\text{spec}_{\text{app}} =$

$(S_{\text{app}}, itr_{\text{job}}, init)$, a node state is defined as a pair $(inb_{\text{next}}, s) \in \text{AANodeState}$ where $inb_{\text{next}} \in \text{MsgBox}$ stores the message for the next period, and $s \in \{\cdot\} \uplus \text{Ready} \uplus \text{Running} \uplus \text{Done}$ represents the current internal state.

The kinds of internal states is as follows. \cdot represents the case that the node is turned off. A **Ready** state is a pair (inb_{cur}, s) of the current inbox inb_{cur} and the initial app state $s \in S_{\text{app}}$ for the job of the coming period. A **Running** state is a pair (sh, itr) of the send history and the running interaction tree $itr \in \text{ITree}_{S_{\text{app}}}$. A **Done** state contains the app state $s \in S_{\text{app}}$ after running the job.

The global step of the abstract asynchronous system is formally described in Fig. 5.3. In a step, each node of the system emits events $E[i]$ and output messages $P[i]$, and then the messages are instantly distributed to their destinations by **accept_msgs**. The formal definition of the node step $- \vdash - \rightarrow_{\text{AANode}} (-, -)$ can be found in the Coq development.

$$\frac{\forall i < |\Sigma|. \quad t_{\text{glob}} \vdash \Sigma[i] \xrightarrow{E[i]}_{\text{AANode}} (\Sigma^1[i], P[i]) \wedge \text{accept_msgs}(t_{\text{glob}}, P, \Sigma^1[i]) = \Sigma'[i]}{(t_{\text{glob}}, \Sigma) \xrightarrow{E} (t_{\text{glob}} + 1, \Sigma')}$$

Figure 5.3 Semantics of abstract asynchronous model

The sketch of the refinement proof is depicted in Fig. 5.4, focusing on the match relation of the incoming messages. We start from a pair of running local states, with the identical send history sh^0 and interaction tree itr^0 . In the abstract side, the node accumulates the incoming messages for the next period in inb_{AA}^0 . In contrast, in the concrete side, the corresponding messages are partitioned into three locations: (1) the global network state N^0 , (2) the inbox inb_{AP}^0 that stores messages arrived before the beginning of this period so that the parsing is already done, and (3) the (unparsed) message packets ps^0 accumulated after the beginning of this period. For this, we define a relation $inb_{\text{AA}} \sim inb_{\text{AP}}, ps$ which implies that messages in inb_{AA} is partitioned into

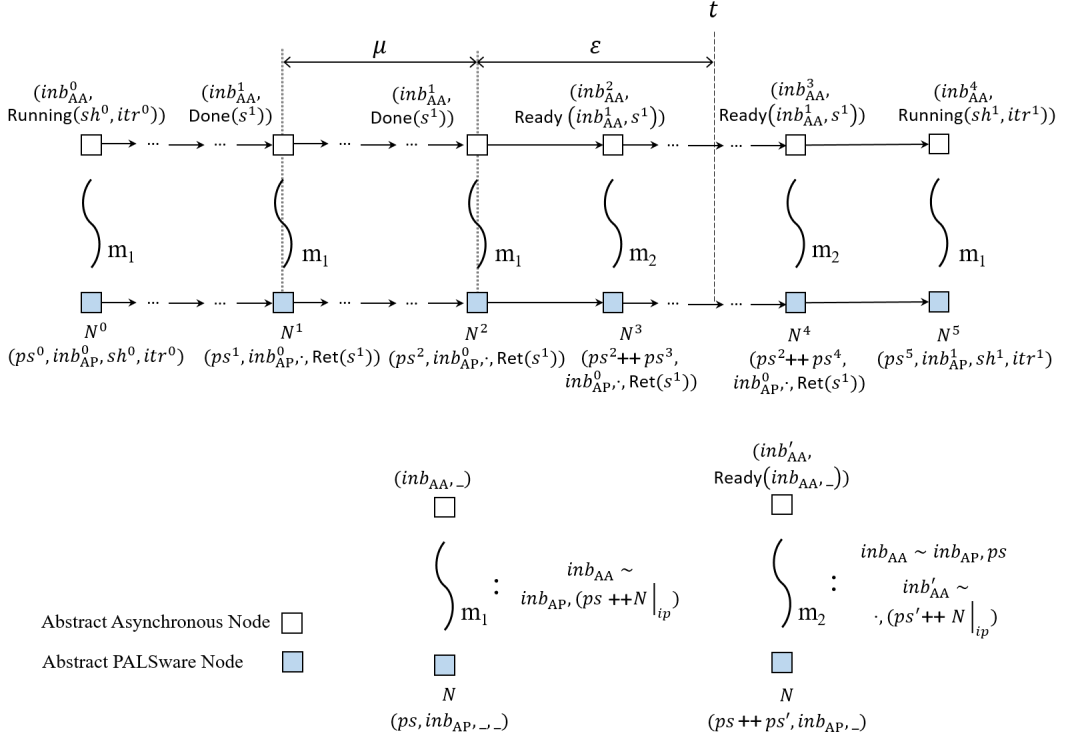


Figure 5.4 Ref 3. Abstract asynchronous model

a message box inb_{AP} and a list of message packets ps . For example, in the first match, the following relation $inb_{AA}^0 \sim inb_{AP}^0, (ps^0 ++ N^0|_{ip})$ holds, where $N^0|_{ip}$ represents message packets in the network N^0 whose actual destination IP address is equal to the IP address of this node ip . In the figure, the curved line with the index m_1 indicates that this relation holds between the two states.

According to the semantics we defined, the job is completed before the time $t - \varepsilon - \mu$ in both of the concrete and abstract system. Then, at the time $t - \varepsilon - \mu$, the abstract side is in the **Done** state, where the match relation m_1 still holds. Since every node in the concrete system has finished its job at this time, no messages are sent until the time $t - \varepsilon$, only after which the next job may begin.

From this, we can deduce that N^2 , the network state at $t - \varepsilon$, is empty so that

the messages of inb_{AA}^1 are only partitioned into inb_{AP}^0 and ps^2 . At this time, the abstract-side node changes its state to **Ready**, copying the inbox inb_{AA}^1 . This **Ready** state means that, the node may initiate the job of the new period any time, with the inbox inb_{AA}^1 and the app state s^1 . The new inbox inb_{AA}^2 stores messages that are arrived right before this step. At the same time, the concrete-side node just keeps accumulating packets.

Now, the two node states are in a new match relation m_2 , which matches each of the two inboxes in the abstract side with the messages of the concrete side. In this case, the accumulated packets in the concrete side can be divided into $ps_2 ++ ps_3$, where ps_3 is newly arrived in this step. While the match relation m_1 at the beginning of the step guarantees $inb_{AA}^1 \sim (inb_{AP}^0, ps^2)$, the newly arrived messages satisfy the relation $inb_{AA}^2 \sim (\cdot, ps^3 ++ N^3|_{ip})$.

Finally, when the job of the new period starts, the concrete-side node parses all the accumulated packets $ps^2 ++ ps^4$. During the process, the messages of ps^2 are merged with inb_{AP}^0 to generate the same inbox inb_{AA}^1 of the abstract side, guaranteed by the match relation m_2 . Now, the resulting inbox is used to produce the initial interaction tree $itr^2 = itr_{\text{job}}(inb_{AA}^1, s^1)$ for the new job in the both of the concrete and abstract nodes. Also, the packets ps^4 are parsed to generate a new inbox inb_{AP}^1 . Message packets arrived later than this time are accumulated in ps^5 . Now the two nodes are in the same situation as the initial match, the simulation will be continued in the rest of the execution.

5.4 Refinement 4: Synchronous Execution

In this stage, we define the abstract synchronous model, in which all job executions and message deliveries are done at the exact synchronization time in a single step. The model takes a step for each nanosecond, since the synchronized behavior removes the need of the atomic time units.

A state of the synchronous model is a pair (t, Σ) , where $t \in \mathbb{N}$ is the global clock time in nanosecond units, and Σ is the list of the synchronous node states. Each node state is a pair (inb, s) defined under an application specification $(S_{app}, itr_{job}, init) \in \mathbf{AppSpec}$, where $inb \in \mathbf{MsgBox}$ is the inbox that stores the messages sent to this node, and $s \in (S_{app})^?$ is the application's state where the `None` value (\cdot) represents the case that the node is turned off.

The formal transition semantics of the synchronous model is shown in Fig. 5.5. In the `WAITSYNCHTIME` case, the period T does not exactly divide the current time t , so it generates empty events and goes to the next state by incrementing the time, without changing node states. On the other hand, the `SYNCHRONIZE` step can be taken if the period divides the current time. In this case, each node takes a local step, and the output messages are instantly distributed to their destinations by `accept_msgs`. Also, the function `AttachTimestamp` attaches the timestamp t to each event in E^0 . The node steps are formally described in Fig. 5.6, where the actual synchronous execution happens in the `RUN` case. In this case, starting from the initial state s_{app} and the inbox inb , the job takes finite steps until it randomly fails or safely terminates.

$$\begin{array}{c}
\text{(WAITSYNCHTIME)} \\
\frac{\neg(T|t) \quad E = \mathbf{repeat}(\cdot, |\Sigma|)}{(t, \Sigma) \xrightarrow{E} (t+1, \Sigma)}
\end{array}
\quad
\begin{array}{c}
\text{(SYNCHRONIZE)} \\
\forall i < |\Sigma|. \quad t \vdash \Sigma[i] \xrightarrow{E^0[i]}_{\text{SNode}} (\Sigma^1[i], P[i]) \wedge \\
\quad \mathbf{accept_msgs}(P, \Sigma^1[i]) = \Sigma'[i] \\
\frac{(T|t) \quad E = \mathbf{AttachTimestamp}(t, E^0)}{(t, \Sigma) \xrightarrow{E} (t+1, \Sigma')}
\end{array}$$

Figure 5.5 Semantics of synchronous model

We explain the structure of the refinement proof of this stage using Fig. 5.7, where the concrete system is the abstract asynchronous model, and the abstract system is the synchronous model. For the period of t_0 , the asynchronous model performs the actual job execution within the time range $(t_0 - \varepsilon, t_1 - \varepsilon - \mu)$. On the contrary, the synchronous model performs the execution in one step at the exact time t_0 .

$$\begin{array}{c}
\text{(INACTIVATED)} \\
\hline
t \vdash (inb, \cdot) \dot{\rightarrow}_{\text{SNode}} ((\text{init_inbox}, \cdot), \cdot) \\
\\
\text{(RUN)} \\
\hline
\frac{t, inb \vdash s_{\text{app}} \xrightarrow{e}_{\text{Period}} (st, outb)}{t \vdash (inb, s_{\text{app}}) \xrightarrow{e}_{\text{SNode}} ((\text{init_inbox}, st), outb)} \\
\\
\text{(ACTIVATE)} \\
\hline
t \vdash (inb, \cdot) \dot{\rightarrow}_{\text{SNode}} ((\text{init_inbox}, \text{init}), \cdot)
\end{array}$$

Figure 5.6 Semantics of synchronous nodes

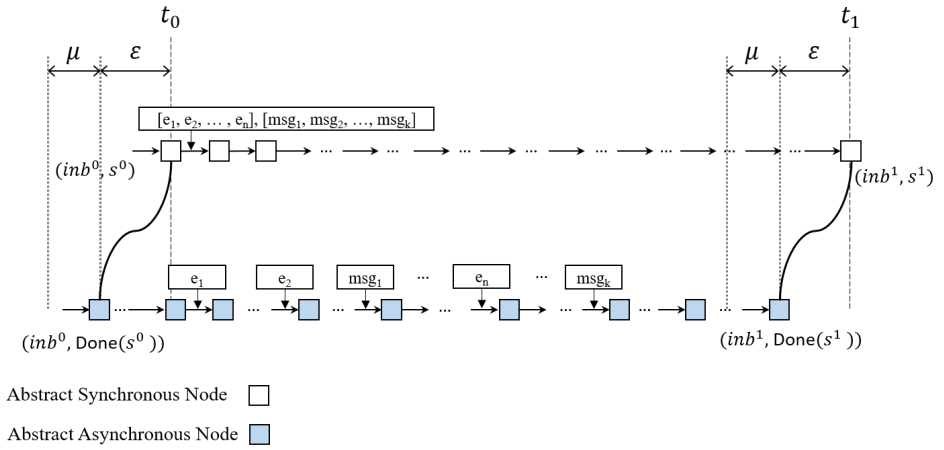


Figure 5.7 Ref 4. Multi-step simulation between asynchronous model and synchronous model

To prove the refinement in this stage, we first define a match relation that matches two equivalent states at the beginning of each period, in order to apply the multi-step simulation technique described in Section 4.1. Specifically, our match relation matches two cases: when the two nodes are turned on and have the same inbox and app state (*e.g.*, as shown in Fig. 5.7), and when the two nodes are both turned off.

Now, we need to show that the concrete node eventually reaches to a state where the accumulated events are simulated by the abstract node and the two nodes are again in the match relation. For this, we advance the concrete system until the time $t_1 - \epsilon$. Then, the accumulated events can be simulated in the one-step execution of

the abstract system, since the job executions in both node basically does the same job: executing $itr_{\text{job}}(inb_0, s_0)$ step by step, with random failures. Then, the abstract system takes silent steps until it reaches to the time t_1 . Since all local tasks in the abstract system do the same thing as the concrete system, the inbox for the next state inb^1 is identical for each node. As a result, the two system states are again in the match relation.

5.5 Refinement 5: Making It Executable

The top-level system model of the framework is the executable abstract synchronous model presented in Section 4.4. This executable model basically works in an analogous way with the (non-executable) synchronous model, except it is written as an interaction tree.

Therefore, the main challenge of this stage only comes from a technical problem about applying the multi-step simulation in a proper way. In particular, the concrete system takes exactly T steps, the length of the synchronization period, for one period. In contrast, the abstract system's number of steps in one period varies depending on each job's interaction tree and app state, since one step is taken for each elimination step of interaction tree, and the job of each task is processed sequentially in the order of task IDs, as shown in the definition of `run.each` in Section 4.4.

We proved the simulation in this way. First, we match the states at the beginning of synchronization, and we let the concrete system take T steps. Then, we need to show that the abstract system can take multiple steps that produces the identical events with the accumulates events from the concrete system's steps, and the states of the two systems are matched again. Since we already has the full history of the concrete system in this period, we can reconstruct the equivalent steps in the abstract system. Obviously, we can easily show that the reconstructed steps produce the identical events, and the resulting states are matched.

Chapter 6

Case Study 1: Active-Standby Resource Scheduling System

Our first case study is a resource scheduling system that is extended from the simple active-standby system. The application system consists of a console task, two controller tasks, and three device tasks. The system assumes that there is a single resource, and only one of the device tasks are allowed to use the resource after acquiring the exclusive ownership.

The controller tasks take the role of scheduler that determines the owner of the resource. Especially, the console and two controllers work in the active-standby mode for reliability; the scheduler service keeps working unless both of the two controllers are not working at an instant.

Although we fix the number of devices as three, the system implementation and the formal verification do not highly depend on the number. The number of devices should be changed without much effort to modify the development.

We believe that this kind of system designs may appear in real embedded systems including shared resources, such as in the case of sharing a single communication

channel to outer environments, or competing for limited electric power.

6.1 High-Level Description

Console A console task is basically the same as the original active-standby system; it takes input from the user and then sends a message to controllers that triggers mode switching between them.

We assume that there is a hardware interface that receives user input in the local environment of this task. For example, it could be a button that the user presses when she wants to switch the active side. When the environment recognizes an input, it stores the signal until the console task calls the `get_user_input` function. If there is a signal, the return value is a nonzero value, namely 1, and otherwise, the return value is zero.

If the task gets a nonzero return value, it sends a *toggle* message to both of the controllers through multicast. The two controller tasks are in a multicast group, so that a message sent to its group IP address arrives at both of them.

Controllers A controller task maintains a queue of device IDs, whose members are waiting for owning the resource, as well as the mode for the active-standby process.

When the job begins for a period, the task first updates its active-standby mode, as shown in Fig. 1.2. If a heartbeat message from the other side of controllers is not arrived, this task always becomes the active mode. Otherwise, it maintains the current mode unless a toggle message is arrived from the console task.

After setting the mode, the task updates the queue according to the messages sent from the devices. Through the messages, a device may request, or release the resource. If the resource is available after the update, the task selects the next owner of the resource. Moreover, the task sends a *grant* message to the new owner, if the task is in the active mode.

When a new owner is selected, the controller sets a timeout value of the ownership, to prevent the system from a stall. If a device node suddenly fails after acquiring the ownership and never recovers, the controller could wait for the release message forever. To prevent this situation, we set a timeout for the ownership with a predefined maximum value. After the maximum number of periods elapses, the controller regards the owner as a failed task and the resource as available.

Finally, the task sends a heartbeat message to the other side of controllers. The message contains the whole information of the current state of this task, so that the other side may copy the state if necessary.

Devices A device task checks whether there is a demand for the resource, from the local environment. If there is a demand, then the task is responsible for getting the resource ownership from the controllers. At the beginning of the job, the task is in one of the three states: idle, waiting, or owning. We are going to explain behaviors of the task for each state.

In the idle state, the tasks checks a demand from outer environment, via an external function of the name `check_demand` that returns an integer value. If it returns a positive integer, it means that the environment requires the resource for that number of periods. Then, the task stores the demand value, and sends an *acquire* message to the controllers. After that, it goes to the waiting state until a grant message arrives. If the return value is zero, it indicates that the environment does not require the resource for now. In this case, the task stays on the idle state.

In the waiting state, the tasks checks its inbox for a grant message. If it does not exist, the state keeps waiting. If there is one, then it immediately goes to the owning state and proceed the job. Note that the task will wait forever unless the controllers send a grant message to this task. For the liveness property, we need to guarantee that a waiting device task will eventually get a grant message.

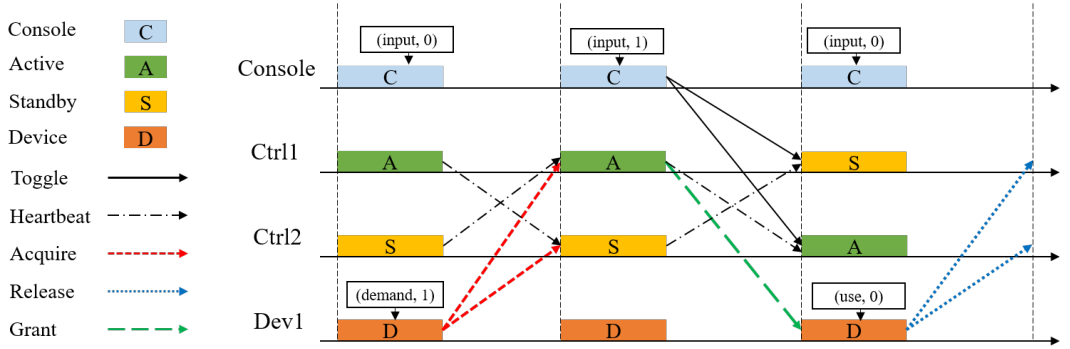


Figure 6.1 An execution of the active-standby system

In the owning state, the task lets the environment use the resource, by calling the `use_resource` external function. After that, it reduces the stored demand value by one. If the value reaches to zero, the task release the resource, by sending a *release* message to the controllers. Finally, it goes to the idle state and completes the job.

Example We explain how the system works by the example of Fig. 6.1, with an emphasis on the resource scheduling. We omit all the device tasks except one, for presentation purpose. In the first period, the device task, say Dev 1, generates an observable event `(demand, 1)`, which means that the external call of `check_demand` returns the integer 1. Now that the device requires the resource, it sends an acquire message to the controllers to ask for the ownership. Then, the controllers receive the message in the next period, and the active side of the controllers sends a grant message to Dev 1 after checking that the resource is available. Finally, in the third period, Dev 1 uses the resource by calling `use_resource`, which is represented as `(use, 0)` in the figure. Since the demand value was one and the task used the resource for one period, it returns the ownership by sending a release message to the controllers.

Note that the active-standby process keeps working simultaneously with the resource scheduling. In each period, the two controllers exchange heartbeat messages

to check each other's state. In the second period, the user requests toggling, which is represented as (input, 1) in the figure. Consequently, the controllers switch their status in the third period.

6.2 Implementation

Now, we explain the concrete C implementation of each kind of tasks, which is slightly simplified and abbreviated for presentation purpose. Refer to the PALSware code in Fig. 2.3 to check the caller of the job functions.

```
1      #include "app.h" // declarations
2      #include "main.h" // pals_send
3      ...
4
5      #define MSG_SIZE 8
6      #define ID_MCAST 6
7      ...
8
9      char TASK_ID = 0;
10     char toggle_msg[MSG_SIZE] = {1};
11
12     void job(inbox_t *inb) {
13         int r = get_user_input();
14         if (r != 0)
15             pals_send(ID_MCAST, toggle_msg);
16     }
```

Figure 6.2 C implementation of the console task

Console The piece of C code in Fig. 6.2 shows the implementation of the console task. The C macro definition of `MSG_SIZE` indicates that the whole application system uses 8-byte messages. Note that this macro is defined for convenience, and PALSware obtains this information from the configuration data declared in `config.h`. Below that, the code sets its task ID as zero, and defines `toggle_msg` as a byte array of the value 10000000. The actual data in the message is insignificant in the system, since

```

1      #include "app.h" // declarations
2      #include "main.h" // pals_send
3      ...
4
5      char TASK_ID = 1; // or 2
6      char grant_msg[MSG_SIZE] = {1};
7
8      char state[MSG_SIZE];
9      ...
10
11     void job(inbox_t *inb) {
12         int tid_owner;
13
14         sync_istate(inb);
15         update_queue(inb);
16         tid_owner = update_owner();
17         if (0 <= tid_owner)
18             pals_send(tid_owner, grant_msg);
19
20         send_hb();
21     }

```

Figure 6.3 C implementation of the controller tasks

the controller only checks the existence of message from the console. The `job` function is the function that PALSware calls for every period.

The `job` function works in the same way as the console in the original active-standby system does. First, it checks whether there has been a user input, by calling `get_user_input`. If the return value is nonzero, then the task sends the toggle message to the multicast group of controllers.

Controllers Next, Fig. 6.3 shows the C code for the controller tasks. The task IDs of the two tasks are set to 1 and 2 in our implementation. Also, the code defines the grant message as 10000000, where the content is again unimportant because the device tasks only checks the existence of message from the controllers. In our implementation, we defined the state as a byte array of size `MSG_SIZE`, in which the

mode, timeout, and a device queue information are encoded. This way makes the construction of heartbeat message trivial.

The `job` function works as follows. The `sync_istate` determines its mode of the active-standby system. From the inbox `inb`, it checks the heartbeat message and the toggle message from the other controller and the console. Then, `update_queue` updates the device queue according to the messages from the devices. After that, `update_owner` checks the availability of the resource, and if it is, it selects the next owner. Note that the selection is only done if the task is in the active mode. If the next owner is selected, the task sends the grant message to the new owner. Finally, it sends its heartbeat message to the other side of controllers, where the message contains its state `state`.

Devices Finally, we explain the code of the device tasks shown in Fig. 6.4. We define the task ID as 3, 4, or 5 for each device task. Also, we define the acquire message `acq_msg` and the release message `rel_msg` as 10000000 and 20000000, respectively so that the controller may distinguish the messages by only checking the first byte. The state of a device task consists of two byte data: `is_owner` and `demand`, where the former data indicates whether it has the resource ownership, and the latter specifies the demand value obtained from the environment. `is_owner` may be one of `UNINIT`, `OWNER`, or `NOT_OWNER`, where the first `UNINIT` value represents a special case that the device is in the first period after turned on. The idle state in the high-level description corresponds to the case in which both `is_owner` and `demand` is zero. The waiting state is when `is_owner` is zero but `demand` is nonzero. Finally, the owning case means that `is_owner` is set as one.

The first part of `job` shows the initialization process of the device task. If the device is just turned on, it initializes the state and send `rel_msg` to the controllers, in order to cancel any previous requests (if exists) from this task, for efficiency. For

example, if the task failed having the ownership before and recovered immediately, the controllers don't have to wait for the maximum timeout until recognizing the failure.

Otherwise, the device first checks whether a grant message arrives, in `sync_dev_state`. In this process, `is_owner` may be changed from `NOT_OWNER` to `OWNER`. After that, it renews the demand from the environment by `update_demand` if there isn't any demand checked before. At last, if the task is the owner, then it lets the environment use the resource by calling `run_device`. After that, if the demand becomes zero, it releases the resource, by sending `rel_msg` to the controllers. If the task is not the owner, it checks the value of `d`, which indicates the new demand value obtained from the environment. If the value is positive, it sends `acq_msg` to the controllers.

6.3 Formally Verified Properties

Our formal verification on the active-standby resource scheduling system is done in two stages. The first one is to prove the correctness of implementation: to write down abstract specifications for each app manually, and prove the simulation relation, to construct a complete application system that provides the parameters and assumptions to the framework. The second one is to prove a desired property of the specification: to show that the abstract model constructed with the abstract specification we wrote can be abstracted further, to a system with a single never-failing controller.

6.3.1 Correctness of Implementation

In this section, we give the (simplified) abstract specifications for the console, controller, and device tasks written as interaction trees. For the simulation proof, the implementation C modules are converted to coq files using the `clightgen` tool, which is included in CompCert. Then, the simulation proof between the specification and

the C implementation is done with the utility lemmas and tactics provided by the VeriPALS framework for this purpose.

The interaction trees in Fig. 6.5, Fig. 6.6, and Fig. 6.7 show the abstract specification for each of the tasks. Comparing this with the implementation shown in Fig. 6.2, Fig. 6.3, and Fig. 6.4, we can see that the specifications' structures are analogous with the implementation, while many concrete details of the C semantics are removed.

In the specification of the console task, `GetUserInput` corresponds to calling the external function `get_user_input()` in C. Also, `SendEvent ID_MCAST toggle_msg` corresponds with calling `pals_send(6, ptr)` where 6 is the actual value of `ID_MCAST` and `ptr` points to a byte array of content 10000000. Note that calling `get_user_input` generates an observable event, where the `pals_send` function is implemented in the PALSware code, which eventually passes the message content to the operating system.

The specification of the controller tasks takes its task ID and the current state as arguments. It needs its own task ID for active-standby operations. When the task is first turned on and no heartbeat messages are received from the other side, the behavior depends on the ID for tie-breaking: one of the tasks becomes immediately active, and other side becomes standby. Also, it uses the task ID to determine the destination of its heartbeat. We delegate the definitions of controller states and the subprocedures `sync_istate`, `update_queue`, and `send_hb` to the Coq development.

The specification of the device tasks is presented in Fig. 6.7. The state `dev_state` has two natural-number fields: `is_owner` and `demand`. Again, the further details are given in the Coq development.

6.3.2 Abstraction to Single-Controller System

For the next step, we proved that the abstract model generated from our hand-written application specifications can be further abstracted to a system with single never-failing controller, under the assumption that the two controllers are never in

the failed states simultaneously.

We interpret this verification as showing two desired properties of the system. The first one is the consistency property that assures that, throughout the whole execution, the cooperation of the two controllers effectively does the scheduling job as if there is a single controller, so that the internal affairs among the console and the controllers are unrecognizable to the clients of the service, which is the device tasks in this case. The second one is the reliability property that the resource scheduling system keeps operating unless the two controllers are simultaneously in the failed states.

For this proof, we define an abstract specification for the single-controller task. Here, we cannot use the specification we defined before, because we should consider a one-period delay that occurs when the active side fails right before sending a grant message. In that case, the standby side recognizes the failure in the next period, and then sends the grant message. To simulate this delay, the new single-controller specification has to take zero or one stuttering period before sending a grant message.

Then, we define a new system with the single-controller specification. This system consists of the single-controller specification, two (silent) dummy tasks that take the slots of the console and one side of the controllers, and the three device tasks. In addition, we encode the assumptions about failures with `Nobehavior` events. Specifically, to rule out the case that two controllers are all in the failed states, we let the system generate `Nobehavior` in such situations.

Finally, we prove the refinement between the previous abstract model and the new model with the single controller. To ease of proof, we first prove the equivalence of the executable abstract synchronous model and the (unexecutable) abstract synchronous model, and then prove the refinement between the systems with the unexecutable model, since the unexecutable is more abstract and support better local reasoning of nodes.

```

1      #include "app.h" // declarations
2      #include "main.h" // pals_send
3      ...
4      #define UNINIT 0
5      #define OWNER 1
6      #define NOT_OWNER 2
7
8      char TASK_ID = 3; // or 4, 5
9      char acq_msg[MSG_SIZE] = {1};
10     char rel_msg[MSG_SIZE] = {2};
11
12     char is_owner;
13     char demand;
14     ...
15
16     void job(inbox_t *inb) {
17         int d = 0;
18
19         if (is_owner == UNINIT) {
20             is_owner = NOT_OWNER;
21             pals_send(ID_MCAST, rel_msg);
22             return;
23         }
24
25         sync_dev_state(inb);
26
27         if (demand == 0)
28             d = update_demand();
29
30         if (is_owner == OWNER) {
31             run_device();
32
33             if (demand == 0) {
34                 is_owner = NOT_OWNER;
35                 pals_send(ID_MCAST, rel_msg);
36             }
37         } else if (0 < d) {
38             pals_send(ID_MCAST, acq_msg);
39         }
40     }

```

Figure 6.4 C implementation of the device tasks

```

1  Definition ID_MCAST: nat := 6.
2  Definition toggle_msg : list bytes := ([1;0;0;0;0;0;0;0])%bytes.
3  ...
4
5  Definition console_spec (inb: list bytes?): itree EvtCall unit :=
6    inp <- GetUserInput ;;
7    (if not (inp == 0) then
8      SendEvent ID_MCAST toggle_msg
9    else Ret ())

```

Figure 6.5 The abstract specification for the console task

```

1  Definition grant_msg : list bytes := ([1;0;0;0;0;0;0;0])%bytes.
2  ...
3
4  Definition controller_spec (tid: nat) (st: ctrl_state) (inb: list bytes?)
5    : itree EvtCall ctrl_state :=
6    st1 <- sync_istate tid st inb ;;
7    st2 <- update_queue st1 inb ;;
8    (st3, tid_owner) <- update_owner st2 ;;
9    (if 0 <= tid_owner then
10     SendEvent tid_owner grant_msg)
11    else Ret ()) ;;
12    send_hb st3 tid ;;
13    Ret st3

```

Figure 6.6 The abstract specification for the controller tasks

```

1  Definition acq_msg : list bytes := ([1;0;0;0;0;0;0;0])%bytes.
2  Definition rel_msg : list bytes := ([2;0;0;0;0;0;0;0])%bytes.
3  ...
4
5  Definition device_spec (st: dev_state) (inb: list bytes?)
6  : itree EvtCall dev_state :=
7    if st.(is_owner) == UNINIT then
8      SendEvent ID_MCAST rel_msg;;
9      Ret (set_owner_status NOT_OWNER st)
10  else
11    st1 <- sync_dev_state inb st ;;
12    (st2, d) <- update_demand st1 ;;
13    (if st2.(is_owner) == OWNER then
14      st3 <- run_device st2 ;;
15      (if st3.(demand) == 0 then
16        st3 <- SendEvent ID_MCAST rel_msg ;;
17        Ret (set_owner_status NOT_OWNER st3)
18      else
19        Ret st3)
20    else
21      (if (0 < d) then
22        SendEvent ID_MCAST acq_msg
23      else Ret ()) ;;
24    Ret st2)

```

Figure 6.7 The abstract specification for the device tasks

Chapter 7

Case Study 2: Synchronous Work Assignment System

Our second case study is a synchronous work assignment system that consists of a single master task who provides works and multiple worker tasks who actually process the works. The master task gets new works to process from its local environment. If there is a new one, the task broadcasts the work to the whole system. Then, every idle worker task who receives the message bids for the work in the next period. Finally, at the beginning of the third period, every one knows who bids for the work, and a predefined selection algorithm known by all tasks picks a worker. The selected worker immediately starts working on it.

This system has an advantage over an asynchronous system in bounding the latency of the assignment to a constant time unless there is no available workers. One could develop an asynchronous version of this system that maintains the list of all workers who is not working, and sends messages to everyone in the list when there is a new work. However, in this case, there is a complication in selecting the worker. If there is no precedence among the workers, the master may assign the work to the

worker whose message arrived first. Otherwise, the master may need a complex algorithm, *e.g.*, waiting for the messages from the workers for a certain length of time, and then choosing a best worker among the workers who sent the message. Also, after choosing the worker, the master may have to announce the result to all the workers participated. On the contrary, our synchronous design has a simple logic in selecting the worker, and also it minimizes the latency of the assignment process.

We think that this kind of systems is realistic in many cases, such as assigning computation tasks to many computing devices, or assigning works to multiple robot agents in the physical world.

7.1 High-Level Description

Master The master task obtains works to process by polling its local environment, via the byte-valued external function `get_new_work`. If the returned byte is nonzero, it means that a new work to process is given, where the byte is its initial data. In that case, the task first generates a fresh work ID, and send to all tasks a *request* message containing the ID and the initial data, through multicast.

In the next period, the master task waits for a period, expecting the worker tasks process the request message.

Then, in the third period, the master task checks *bidding* messages from the workers. If there is at least one such message, it means that there is a worker who is going to process the work. In this case, the master task goes back to the first stage and start polling for a new work. If there is no bidding messages, it means that currently there is no worker task available for the work. Then, the master sends the request message again, until a worker task becomes available.

Workers When a worker task has no work to process, it remains idle until a request message is arrived from the master. If there is one, the idle task sends a bidding

message to all tasks through multicast. Then, in the next period, every task knows who is bidding for the work. We assume that the selection algorithm is known to all the worker tasks. For example, our implementation adopts the rule that a task with a greater task ID gets a higher priority. Therefore, among the bidders, the one with the highest priority immediately starts processing the data, while other workers remain idle.

Once a worker task is assigned a work, it processes the work over several periods. Calling the external function `do_work(i, d)` with the work ID i and the current work data d lets the environment process the work. After processing, the function returns the next data. If the data is zero, it means the work is done. Otherwise, the task stores the data and keeps processing it in the next period.

Example A concrete example of Fig. 7.1 describes how the system works. The system in the figure has three worker tasks, where Worker 3 has the highest priority and Worker 1 has the lowest. In the first period, Worker 2 already has a work of ($ID=id_1$, $state=y_0$). So, Worker 2 continues to work on the work by calling the “work” external function that returns the next work state y_1 . At the same period, Master gets a new work of an initial data x_0 , then it generates a new id id_1 and broadcasts a request message ($ID=id_1$, $state=x_0$). Then, in the second period, each of the two idle workers, Worker 1 and Worker 3, bids for the new work by broadcasting a bid message. Meanwhile, the master task waits until the next period for the bidding result. Finally, everyone agrees that Worker 3 has the highest priority among the participants, so Worker 3 immediately starts working, while Worker 1 goes back to the idle state and Master starts polling for a new work again. If there were no bidders, Master would repeat sending the request message until the assignment is done.

7.2 Implementation

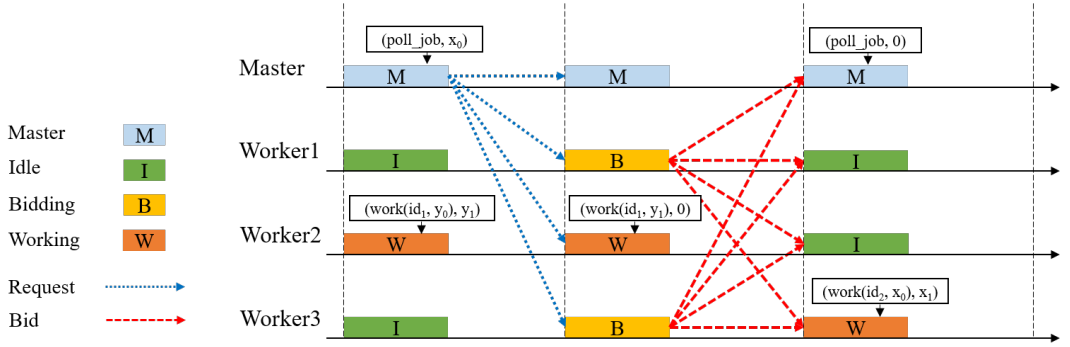


Figure 7.1 An execution of the work assignment system

Master The main part of C implementation of the master task is shown in Fig. 7.2. The job is composed of three stages: `update_status`, `poll_env`, and `send_request`. We are going to explain the code in the following paragraphs.

The first `update_status` stage changes its status. If the current status is `NEW_WORK`, it implies that the master task sent request messages in the previous period, so it enters to the `PENDING` status. If the current status is `PENDING`, the task searches the inbox for a bidding message. If it succeeds to find one, the status is changed into `IDLE`, since the work assignment process is completed. Otherwise, the status goes back to `NEW_WORK`, in order to send the request messages again.

Next, `poll_env` runs only when the task is idle. It polls the environment by calling `get_new_work`. If it returns a nonzero data, the master task prepares for sending request messages for the new work. Specifically, it changes the status into `NEW_WORK`, assigns a new work ID, and stores the data.

Finally, `send_request` actually sends request messages for the new work, if it exists. It writes the ID of the new work and the data in the message, and call `pals_send` to send the message to all tasks.

Workers The C code for the worker tasks is shown in Fig. 7.3. This code is composed of three main procedures shown in the code, `check_bidding`, `check_new_work`, and `continue_work`.

First, if the status is `BIDDING`, it checks whether it succeeds to get the work, where `is_bidding_succ` does the job. In our implementation, it checks whether its task ID is the greatest among workers who sent bidding messages.

After that, if the status is `IDLE`, the task checks the inbox to see whether there is a new work requested by the master. The function `find_request` searches the inbox, and if there is a new request, it stores the message to `r` and returns a nonzero value. Then, the worker participates in bidding. It changes its status into `BIDDING`, stores the request information, and sends the bidding message `bid_msg` to all tasks.

The final branch is taken if the status is `WORKING`. In this case, the worker lets its local environment process the current work by calling the `do_work` external function. If the returned data is zero, it means the process is done, so the status is changed to `IDLE`.

```

1  #include "app.h" // declarations
2  #include "main.h" // pals_send
3
4  #define MSG_SIZE 2
5  #define IDLE 0
6  #define NEW_WORK 1
7  #define PENDING 2
8  ...
9
10 char TASK_ID = 0;
11 char req_msg[MSG_SIZE] = {0, 0};
12
13 char status;
14 char next_work_id;
15 char data;
16 ...
17
18 void update_status(inbox_t *inb) {
19     if (status == NEW_WORK) {
20         status = PENDING;
21     } else if (status == PENDING) {
22         if (find_bidding(inb))
23             status = IDLE;
24         else
25             status = NEW_WORK;
26     }
27 }
28
29 void poll_env() {
30     char d = get_new_work();
31     if (0 < d) {
32         status = NEW_WORK;
33         ++ next_work_id;
34         data = d;
35     }
36 }
37
38 void send_request() {
39     req_msg[0] = next_work_id;
40     req_msg[1] = data;
41     pals_send(ID_MCAST, req_msg);
42 }
43
44 void job(inbox_t *inb) {
45     update_status(inb);
46     if (status == IDLE) poll_env();
47     if (status == NEW_WORK) send_request();
48 }

```

Figure 7.2 C implementation of the master task

```

1  #include "app.h" // declarations
2  #include "main.h" // pals_send
3
4  #define IDLE 0
5  #define BIDDING 1
6  #define WORKING 2
7  ...
8
9  char TASK_ID = 1; // or 2, 3, ..., 8
10 char bid_msg[MSG_SIZE] = {1, 0};
11
12 char status;
13 char work_id;
14 char data;
15 ...
16
17 void check_bidding(inbox_t *inb) {
18     if (is_bidding_succ(inb))
19         status = WORKING;
20     else
21         status = IDLE;
22 }
23
24 void check_new_work(inbox_t *inb) {
25     char r[2];
26     if (find_request(inb, r)) {
27         status = BIDDING;
28         work_id = r[0];
29         data = r[1];
30         pals_send(ID_MCAST, bid_msg);
31     }
32 }
33
34 void continue_work() {
35     data = do_work(work_id, data);
36
37     if (data == 0)
38         status = IDLE;
39 }
40
41 void job(inbox_t *inb) {
42     if (status == BIDDING)
43         check_bidding(inb);
44
45     if (status == IDLE)
46         check_new_work(inb);
47     else
48         continue_work();
49 }

```

Figure 7.3 C implementation of the worker tasks

Chapter 8

Results

This chapter summarizes the result of this work, with experimental results for the testing performance. First, we present how the development is structured in terms of the lines of code in each part. Then, we present our experimental results, to compare the performance of the three testing procedures that we support. We conducted the experiments on our two case-study application systems.

8.1 Development

Our development is a mixture of C implementation and Coq formal definitions and proofs. In the C development, we developed our version of PALSware and the two application system implementations for case studies. In the Coq development, we defined the formal models, as well as the formal refinement proofs between the models. Moreover, it contains the formal verification result for the resource-scheduling application system.

Table 8.1 shows the lines of code information for each part of the C development. The total number of lines of code in the C development is 1608 lines, which are

	Framework				ResSched			WorkAssn			
	Infra	Impl	Test	Sum	Impl	Test	Sum	Impl	Test	Sum	Total
LoC	135	306	373	814	390	98	488	225	81	306	1608

Table 8.1 Lines of code in the C development

partitioned into three parts: the framework part and the two application systems.

The framework development is written in 814 lines total. Among them, the Infra column represents the code piece that implements the API of our operating system model, from the libraries provided by the Linux system (*e.g.*, `timerfd` API for implementing timer services). This part is the trusted computing base of our work; we trust that the Linux libraries and this Infra code is properly modeled in our operating system model. Next, the Impl column represents the implementation of PALSware. This part is the verification target of our Coq proof. Then, the Test column represents the infrastructure for testing. Specifically, it implements the interface function for the direct linking with OCaml. The code length is relatively long, since the code is responsible for correct conversions between C data structures and OCaml data structures back and forth, and sometimes it requires complex programming patterns.

The ResSched columns correspond to C development of the resource scheduling active-standby system, which is written in 488 lines of code. Among them, 390 lines account for the implementation of the tasks in the system. The rest 98 lines are for linking with OCaml, where most of the code is boilerplate.

The WorkAssn columns correspond to C development of the synchronous work assignment system, which is written in 306 lines of code. Among them, 225 lines account for the implementation of the tasks, and the rest 81 lines are boilerplate code for linking with OCaml.

The lines of code information for the Coq development is shown in Table 8.2, whose total length is 62131 lines. The code is divided into two parts: the framework and the application system verification of the resource scheduling system.

	Framework						ResSched			
	Lib	Mdl	Thr	ImplV	RefPf	Sum	ImplV	AbsV	Sum	Total
LoC	5397	9695	5326	12712	15636	48716	9986	3379	13365	62131

Table 8.2 Lines of code in the Coq development

The total 48716 lines of Coq development for the framework is partitioned into six categories. First, the Lib column represents the Coq utility library that we wrote or imported from other open-source projects. For example, it contains useful definitions for list structures, and proofs about their properties. Next, the Mdl column corresponds to the formal definitions for system models, and brief proofs about their properties. The Thr column represents our general theory about simulation and refinement, *e.g.*, the multi-step simulation and its adequacy proof. The ImplV column represents the actual verification of the concrete C implementation of PALSware. The code length is quite long since it deals with the complex C formal semantics. At last, the RefPf column shows the number of lines for the refinement proofs between system models.

The ResSched columns accounts for our verification result of the resource scheduling application system. Of the total 13365 lines, 9986 lines are devoted to the verification of the implementation, *i.e.*, each simulation proof between a task implementation and its specification. The rest 3379 lines corresponds to the further abstraction proof, which shows that the system can be abstracted to a never-failing single-controller system.

8.2 Experimental Results

We conducted experiments to compare the testing performance of our three testing methods. To recall, we are going to briefly describe the three methods. The first method is to link the C code of the application system directly to the generic abstract model, using the OCaml foreign function interface functionality. The second method

is to use our C-to-ITree conversion function to automatically generate the interaction trees from the application system implementation. The above two method is available even if the hand-written specifications for the tasks are absent. The last one is to build the abstract model with the manually written specifications for the system’s tasks in Coq, and extract it to an OCaml program. This method is only applicable when the specifications are provided by the user.

The experiments are done on an Intel I7-7500U CPU (2.70GHz, 2 cores) machine with 16GB RAM that runs Ubuntu 18.04 LTS via the Windows Subsystem for Linux on Windows 10. The Coq version is 8.13.2 and the OCaml compiler version is 4.12.0. Also, all the time data in the experiment is the ‘real’ time measured by the `time` command of the Z shell.

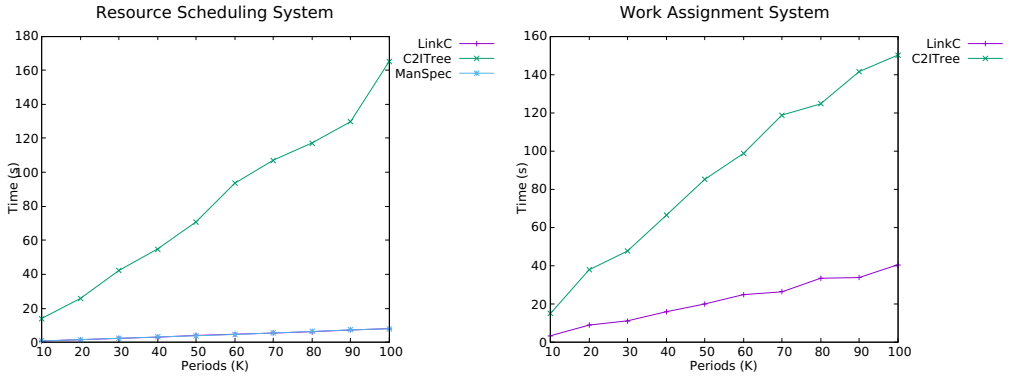


Figure 8.1 Experimental results comparing performance of the three testing methods

The graphs in Fig. 8.1 shows the correlation between the testing time and the number of periods in a single test run, for the two application systems. In the graphs, the labels LinkC, C2ITree, and ManSpec represent data from the first method that links C with OCaml, from the second method that uses the C-to-ITree conversion, and from the third method that uses manually written specifications, respectively. Each data point is obtained by computing the average of 10 runs.

The left graph shows the results for the resource scheduling system. We can see

that the C2ITree method is much slower than the other two methods, which shows very similar performance with each other. Fitting a line $y = ax$ to the data plots, C2ITree fits to $a = 1.515$, which implies that the testing speed is equivalent with running the actual system with the synchronization period length of 1.5 milliseconds. On the other hand, the other two method fits to the linear curve of slope $a = 0.079$, which matches to the actual system with the period length of 79 microseconds. In summary, the C2ITree method is about 20 times slower than the other two methods in this system.

Besides, the right graph shows the results for the work assignment system. In this graph, the slope of C2ITree is 1.590 and that of LinkC is 0.397. In this system, the C2ITree method is about 4 times slower than the LinkC method. Note that the graph does not have ManSpec data, since we do not provide manual specifications for this system.

As a cause of the slow performance of the C2ITree method, we suggest its interpreter-like execution process. The conversion function from C to interaction tree constructs an interaction tree that simulates the execution proces of C programs that conforms to the Clight formal semantics. The converted interaction tree carries a Clight state which contains a memory model state, where the memory model is extracted in OCaml as a tree data structure of byte lists. Also, evaluating expressions is done following the procedure formally defined in Clight, instead of using the native evaluation process.

Chapter 9

Related Work

There have been several approaches related to our research goal, which is to achieve high-level safety guarantee of cyber-physical systems. We categorize them into three groups for each topic. In the first topic, we review the former studies about the PALS pattern itself and the PALSware implementation and verification efforts. Second, we compare ours with existing distributed system verification frameworks that supports developing user application systems. Finally, we discuss studies about verifying C programs.

9.1 PALS Pattern and PALSware Verification

PALS Architectural Pattern The physically-asynchronous logically-synchronous (PALS) architectural pattern is originally designed in UIUC[9, 10] to correctly implement synchronous distributed systems on asynchronous environments, taking non-deterministic factors into account. PALS is designed as a replacement of globally-asynchronous logically-synchronous (GALS) architecture in avionics systems, with the advantage of lower complexity that comes with globally synchronous system de-

signs. In the original design of PALS, the sender task is responsible for sending a message not too early; it assumes that the sender never transmits messages within a certain time after the period begins.

There is an approach to applying formal verification to PALS systems[11]. In this work, the authors define a formal PALS model as a transformation from a synchronous system model, in the Real-Time Maude tool, and verified the correctness of the transformation by showing the bisimulation between the original synchronous model. However, their achievement differs from ours, in the aspect that our work verifies the **implementation** of PALSware and application system, where this work verifies simplified and abstract models of systems.

PALSware This PALS pattern is implemented as middleware for real-time distributed systems, called PALSware[7]. This separation of concerns helps system developers by allowing them to assume a synchronous environment in designing real-time distributed systems. The PALSware design includes several practically useful functionalities that our work didn't include, such as fault managers, end markers for atomicity, and multi-phase and multi-rate system extensions. A model checking approach has been applied[8] to the PALSware implementation. This work uses the CBMC tool to prove three properties of the message service of PALSware, and then combines this result with additional model checking on application systems. Comparing to this work, our verification result guarantees the full **functional correctness** against the system's synchronous behaviors. Also, the model checking approach often fails to give results when it encounters state explosions.

9.2 Verification Frameworks for Distributed Systems

There have been several research projects that aim to build a general formal verification framework for distributed systems. This section introduces such projects one by

one, and compares them with our work.

Verdi Verdi[20] is a framework for implementing and verifying distributed systems in Coq. To use this framework for building a distributed system, the system developer first implements each distributed task of the system in Coq. Then, the Coq’s extraction mechanism generates executable OCaml programs for each task, which can be deployed as an actual system with the runtime library provided by Verdi. For verification, Verdi supports various kinds of network models to specify the system’s behaviors, from the reliable semantics which only permits reordering of packets to a network semantics which permits packet loss and duplication. On one of the network models, the user can verify formal properties of the system’s behaviors. As a case study, they verified the Raft consensus algorithm to demonstrate effectiveness of the framework.

Notably, Verdi provides a verified system transformer that converts a system built on one network model to a new system on another network model, by augmenting the input system with proper handlers that detect and resolve a certain kind of faults. In contrast, our network model only supports the reliable network behaviors, since the correctness of PALSware depends on it.

On the other hand, our network model properly supports real-time reasoning, which is crucial for verifying PALSware. Also, compared with this work, we designed a detailed model for a real-time operating system to reduce the conceptual gap between the real world and our mathematical reasoning.

Another difference is that our system supports C programs, which is, from our perspective, more realistic in implementing real-time embedded distributed systems. Using C programs for application systems definitely makes the formal verification difficult, so verification experts are needed in verifying application systems. Nevertheless, our framework helps the verification by supporting modular reasoning for

each distributed task. Moreover, our framework also serves as an efficient verified testing tool. For the cases that conducting the formal verification seems hard, thorough testing on the application system is possible on our framework, instead of formal proofs.

IronFleet IronFleet[21] is another verification framework for distributed system, based on the Dafny language. An advantage of this framework is that it utilizes an SMT solver in the verification process, which greatly helps in automating proofs. Their network model is not restricted to the reliable network assumption.

As Verdi does, this framework lacks support for real-time verification, and it requires the developer write programs in their own language. Also, while their reduction technique greatly ease the proof effort, its correctness is not integrated in the mechanized code base.

To sum up, existing verification frameworks for distributed systems are good at dealing with various network assumptions or reducing proof efforts, while our network model enables real-time reasoning which makes the verification of PALSware possible. Moreover, on our framework, the system developer can rely on the synchronous environment assumption and implement their system in the C language.

9.3 Verifying C Programs

Formally verifying C programs is one of the major branches of formal system verification. The CompCert project[12], a realistic C compiler fully verified in Coq, designed formal C semantics that conforms to the C standard, and also formal translations from C to assembly. Then, it is followed by many formal verification projects that are based on the CompCert’s results. CertiKOS[13] is a concurrent operating system kernel whose functional correctness on the Clight formal semantics is verified in Coq with their certified abstract layer approach. Verifiable C of VST[14] is a C program

verification tool. It can be seen as a separation logic prover for C programs, also based on the Clight semantics. CompCertM[15] is an extension of CompCert that fully supports separate compilation. In their work, the authors showed that their proof technique called RUSC is applicable to verifying programs.

Among the verification efforts, a study on verifying a web server[22] is comparable to our work. In this work, the authors implemented a simple ‘swap’ web server, and wrote its abstract specification as an interaction tree, like we did in our work.

However, this work differs from our work in proving the relation between the implementation and specification. This work uses the VST tool to prove a hoare triple that indirectly implies the refinement between the implementation and specification, but it doesn’t give a further soundness proof of it. As a result, the final theorem is partitioned into two unintegrated parts: one proving the hoare-triple relation, and the other proving that the specification shows a desired property. On the other hand, we directly defined execution models for both of the C implementation and the interaction-tree specification, and proved the refinement between them. Therefore, our subproofs are well-composable with each other to prove the final refinement theorem. Additionally, while this work verifies a single server’s behaviors with multiple client models, our work verifies the whole-system implementation.

Chapter 10

Conclusion and Future Work

In this work, we developed the VeriPALS framework for implementing and verifying distributed systems with synchronous designs written in C. The framework includes an implementation of PALSware, which provides a virtually synchronous environment to the upper application layer, on a physically asynchronous environment of network and real-time operating systems. Since developing systems in synchronous designs greatly reduces the system complexity, PALSware is well suited for developing safety-critical systems. The PALSware is formally proven in Coq that an arbitrary application system built on it refines the system's ideal synchronous behaviors, as long as the execution times of each job are bounded by a certain time limit.

For system verification, our framework combines the user-given local simulation proofs between each app implementation and its specification to prove the global refinement between the real-world model and the ideal abstract synchronous model. The user may conduct further formal verifications for proving that a desired property holds for the abstract model's behaviors. This way is much simpler than proving that a property holds for the real-world model's behavior, since the real-world model

contains concrete and detailed components in it, such as formal C semantics and operating system services.

Moreover, the framework serves as an efficient testing tool for application systems. For testing, our abstract synchronous model can be extracted to a single executable OCaml program that simulates the actual distributed system. In terms of resource and time, using this tool is more efficient than deploying the actual distributed system for testing. We provide three testing methods, in which only one method requires manually written specifications from the user.

To demonstrate the effectiveness of the VeriPALS framework, we conducted two case studies of application systems. The first system is a resource scheduling system with two controllers that operate as an active-standby system for reliability. For this system, we applied formal verification to prove the correctness of implementation against manually written specifications. Additionally, we proved that the system can be abstracted further, to a system with a single, more reliable controller. The second system is a synchronous work assignment system that consists of a single master task and multiple worker task. For the two systems, we applied the three testing methods and compared their performances.

Future Work In the real-world model of the framework, the operating system model runs a C program for its application program. We believe that our work can be easily integrated with the CompCert’s verification results, to replace the application program from C to assembly. This work would reduce the gap between our model and the actual system, since the physical hardware runs according to the compiled assembly program. Also, we could prove that compiling our system using CompCert preserves the system’s behavior, which gives a higher safety guarantee.

Another future direction is to extend our PALSware implementation. As we clarified, the current version is a simplified version that does not support transmitting

multiple messages to a single task in one period. Also it does not support multi-phase and multi-rate extensions introduced in the original work of PALSware. We believe we can generalize the current implementation to remove those restrictions.

Finally, we plan to apply the framework to develop realistic cyber-physical systems, such as autonomous driving systems or underwater vehicles. From this, we hope to contribute to achieving high-level safety assurance for safety-critical computer systems in the real world.

Bibliography

- [1] Gawand, H. Laxman, A. K. Bhattacharjee, and K. Roy. “Securing a cyber physical system in nuclear power plants using least square approximation and computational geometric approach,” *Nuclear Engineering and Technology* 49.3 pp. 484-494, 2017.
- [2] K. Sampigethaya and R. Poovendran. “Cyber-physical system framework for future aircraft and air traffic control,” in *2012 IEEE Aerospace Conference. IEEE*, 2012. pp. 1-9.
- [3] B. Chen, Z. Yang, S. Huang, X. Du, Z. Cui, J. Bhimani, X. Xie, and N. Mi, “Cyber-physical system enabled nearby traffic flow modelling for autonomous vehicles,” in *2017 IEEE 36th international performance computing and communications conference (IPCCC)*, Dec 2017. pp. 1-6.
- [4] U. S. – Canada Power System Outage Task Force, “Final report on the august 14, 2003 blackout in the United States and Canada”, Washington, WA, Apr. 2004
- [5] European Space Agency, Ariane 501 Inquiry Board Report, <https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>

- [6] A. Albee, S. Battel, R. P. Brace, G. Burdick, J. Casani, J. S. Lavell, C. Leising, D. Macpherson, P. Burr, and D. Dipprey, “Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions,” 2000.
- [7] A. Al-Nayeem, C. Kim, W. Kang, P. L. WuP, and L. Sha, “Middleware design for physically-asynchronous logically-synchronous (pals) systems,” in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2013. pp. 1-10.
- [8] M. Y. Nam, L. Sha, S. Chaki, and C. Kim, “Applying software model checking to PALS systems,” in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)* October 2014. pp. 5B4-1.
- [9] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Olveczky, “PALS: Physically asynchronous logically synchronous systems,” 2009.
- [10] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, and D. D. Cofer, “A formal architecture pattern for real-time distributed systems,” in *2009 30th IEEE Real-Time Systems Symposium*, December 2009, pp. 161-170.
- [11] J. Meseguer and P. C. Ölveczky, “Formalization and correctness of the PALS architectural pattern for distributed real-time systems,” in *International Conference on Formal Engineering Methods*, Springer, Berlin, Heidelberg, 2010. pp. 303-320.
- [12] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, 52.7, pp. 107-115, 2009.
- [13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, “Certikos: An extensible architecture for building certified concurrent OS kernels,”

- in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016. pp. 653-669.
- [14] A. W. Appel, “Verified software toolchain,” in *European Symposium on Programming*, Springer, Berlin, Heidelberg, March 2011. pp. 1-17.
 - [15] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C. K. Hur, “CompCertM: CompCert with C-assembly linking and lightweight modular verification,” in *Proceedings of the ACM on Programming Languages, (POPL)*, 2019, pp. 1-31.
 - [16] L. Y. Xia, Y. Zakowski, P. He, C. K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, “Interaction trees: representing recursive and impure programs in Coq,” in *Proceedings of the ACM on Programming Languages, (POPL)*, 2019, pp. 1-32.
 - [17] PTP daemon, <http://ptpd.sourceforge.net>
 - [18] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, “CompCertTSO: A verified compiler for relaxed-memory concurrency,” in *Journal of the ACM (JACM)*, 60(3), 2013. pp. 1-50.
 - [19] C. K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis, “The power of parameterization in coinductive proof,” in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, January 2013. pp. 193-206.
 - [20] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, “Verdi: a framework for implementing and formally verifying distributed systems,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2015. pp. 357-368.

- [21] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, “IronFleet: proving practical distributed systems correct,” in *Proceedings of the 25th Symposium on Operating Systems Principles* October 2015. pp. 1-17.
- [22] N. Koh, Y. Li, Y. Li, L. Y. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic, “From C to interaction trees: specifying, verifying, and testing a networked server,” in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, January 2019. pp. 234-248.
- [23] The VeriPALS Coq Development, <https://github.com/kim-yoonseung/pals-thesis-dev>

초록

사이버 물리 시스템의 안전성을 높이는 일은 항상 중요한 연구 주제가 되어왔다. 그 이유는 많은 사이버 물리 시스템이 안전 우선 시스템이기 때문인데, 이는 실제 시스템 구동 중에 오류가 발생할 경우 큰 사고로 직결될 수 있음을 의미한다. 더욱이, 사이버 물리 시스템이 가지는 실시간성, 분산성이 시스템의 복잡도를 높여 위험성을 증가시키므로 안전성을 높이는 일은 매우 중요하다.

시스템의 복잡도 문제를 해결하기 위해, PALSSware라는 미들웨어가 고안되었다. 이 미들웨어는 비동기식으로 동작하는 네트워크와 운영체제 환경 위에서 가상의 동기식 환경을 애플리케이션 층에 제공하는 역할을 한다. PALSSware를 사용하면 시스템을 동기식 환경에서 디자인할 수 있게 되어, 시스템의 복잡도를 크게 낮추는 것이 가능해진다.

하지만, PALSSware에 버그가 있을 경우 그 악영향이 매우 크게 나타날 수 있다. 우선 이 미들웨어를 사용하는 모든 애플리케이션 시스템에 버그가 존재하게 된다. 또한, 미들웨어의 버그를 찾는 일은 일반 프로그램의 버그를 찾는 것보다 매우 어려운 문제가 될 수 있다.

이 문제를 해결하기 위해, 우리는 VeriPALS라는 프레임워크를 개발하였다. 이 프레임워크는 수학적으로 엄밀하게 검증한 PALSSware의 C 구현체를 포함하고 있어 안전한 시스템 구현을 돕는다. 또한, 애플리케이션 시스템을 Coq 위에서 수학적으로 엄밀히 검증할 수 있는 기능을 지원한다. 더 나아가서, 이 프레임워크는 실행 가능한 모델을 효율적인 랜덤 테스트 툴로서 제공한다. 우리는 이 프레임워크 위에서 두 종류의 애플리케이션 시스템을 개발하고 테스트 및 엄밀 검증하여 이 프레임워크의 유용성을 보였다.

주요어: 정형 검증, 분산 시스템, 실시간 시스템, 동기식 시스템, 정리 증명

학번: 2013-23107

Acknowledgements

I would like to thank my supervisor, Prof. Chung-Kil Hur, for directing my long journey with insights and inspirations. Prof. Kwangkeun Yi's helpful comments always prevent me from going out of the right way, throughout the whole period of my PhD training. Prof. Cheolgi Kim has introduced a great research topic to me. I'm grateful to Prof. Chang-Gun Lee for examining my thesis with helpful comments. Prof. Jeehoon Kang, a former colleague I used to rely on, is now making a successful career.

I thank Youngju, Juneyoung, and Yonghyun for working with me in many successful projects, and for having a nice time with me. It was lucky to work with Sung-Hwan on PALSware verification. Working with Minki and Dongjoo was a great experience. I remember Sanghoon and Dongyeon were not only good colleagues but also nice guys. Hanging out with Dongkwon and other ROPAS members was always joyous. I hope new members have a good time here.

My PhD work is supported by The Korean Foundation for Advanced Studies. With their help I was able to solely concentrate on my research.

Without the encouragement and emotional support from my parents and sister, I would never finish this PhD work. Finally, I want to give special thanks to Ring, who came to me one day in November and is now my only shelter in this world.