Sixteen Years of Formalization for Embedded Real-Time Operating Systems in PowerEpsilon

Ming-Yuan Zhu
CoreTek Systems
11th Floor, 1107B, CEC Building
6 South Zhongguancun Road
Beijing 100086
People’s Republic of China
E-Mail: ZHUMY@CORETEK.COM.CN

This article is dedicated to the 16th anniversary of CoreTek Systems.

Abstract
This paper provides a summary and review of my work on formalization of embedded real-time operating systems in mathematical theorem proof development system PowerEpsilon during the past sixteen years. We started from providing a formal specification and analysis of an embedded real-time operating system kernel called δ-CORE based on a language-driven approach followed by the formalization of a partition-based RTOS kernel, a monitor-based RTOS kernel and a virtual machine based RTOS kernel. We then introduced the concept of operating system machine in order to model the interrupt handling mechanism. We are also be able to formalize the software development process for RTOS. Other concepts and features have been modeled in the work include the security policy, the BSPs and device drivers, the RTOS for multi-core processors, and the software development environments for embedded RTOSes.

1 Introduction
This paper is a summary and review of my work on formalization of embedded real-time operating systems in mathematical proof development system PowerEpsilon during past sixteen years.

1.1 About PowerEpsilon
PowerEpsilon is my personal creation. It is a strongly-typed polymorphic functional programming language based on Martin-Löf’s type theory and the calculus of constructions. In PowerEpsilon, a scheme for inductive define types are introduced. The system has been implemented using the software development system AUTOSTAR constructed by author. PowerEpsilon is a proof checker much similar to other mechanical proof checkers, such as ACL2, LCF, Isabelle, Nuprl and HOL, which are completely formal user-controlled systems. However, PowerEpsilon is more powerful than LCF and Nuprl, in which the equality and induction rules for arbitrary inductive types are definable.

1A revised version of the paper with the same title is submitted to ACM SIG Operating Systems Review.
1.2 About DeltaSystem

The formalization work for operating systems was started in 2000 when I created CoreTek Systems with a number of partners. CoreTek Systems is a company specialized on research and development as well as marketing of embedded real-time operating systems, integrated development tools, testing tools and application software components. Our operating system product family is named DeltaSystem. The initial product package DeltaOS includes DeltaCORE - a real-time operating system kernel, DeltaNET - a networking sub-system, DeltaFILE - a file system, DeltaGUI - a GUI package and so on. We have also a product called DeltaSVM which is a hypervisor based RTOS specially designed for control-oriented applications, a product called ACoreOS which is a partition based RTOS jointly developed with AVIC and specially designed for avionics-oriented applications, a product called DeltaDSP which is a RTOS for digital signal processors, a time-triggered real-time kernel DeltaTT and a real-time kernel with small footprints called DeltaSCC. We have used the following naming rules for our products: prefix with $\delta$ for embedded software components including the RTOS kernels, prefix with $\lambda$ for development tools and prefix with $\gamma$ for testing tools. I am very proud to establish this naming system. In this article, we will selectively use both $\delta$-$X$ and Delta$X$ for the same thing.

During the past sixteen years, I have never written a line C code for our products. My duty are management and decision making. My way to understand a theoretical concept or to find a technical solution is to establish a formal model\[38\]. It has never failed.

2 The Language Driven Approach for Formal Models of DeltaCORE

We propose a practical method for the specification and verification of operating systems. We will start the investigation by defining a high level language, the specification and verification methods are given based on the denotational semantics of the language. We want to have a formalism suitable for all concurrent systems.

2.1 Formal Specification of DeltaCORE

My starting point on formalization of DeltaCORE was actually a reverse engineering work. When I started, the DeltaCORE was already delivered for sales on market. The formal specification of DeltaCORE was used internally for purpose of deeper understanding and better documentation of DeltaCORE. The result has been summarized as a technical report “Formal Specifications of Operating Systems – A Language Driven Approach”.

The work was based on a language driven approach. We defined a subset of C language together with the system service routines of DeltaCORE called ToyC. ToyC can be viewed as a concurrent programming language with C-style syntax and system service routines of DeltaCORE. The semantics of DeltaCORE is then defined as the semantics of ToyC. The abstract syntax, static semantics and dynamic semantics of ToyC were defined in terms of PowerEpsilon.

The following properties of DeltaCORE were investigated:

- The correctness analysis of the system. A revised version of this part of work entitled “A Provably Correct Operating System: $\delta$-CORE” appeared in ACM SIG Operating Systems Review, Volume 35, Number 1, January, 2001[35].

- The complexity analysis of system service calls (APIs). This part of work is summarized in an unpublished manuscript entitled “The Proof of Deterministics of $\delta$-CORE”. A consequence of this part of work is that the system service calls for deletion of semaphores and message queues are non-deterministic. Therefore the usage of these two functions must be used with care. Another consequence from the analysis is that a direct implementation of priority-based task queues should be the double-linked lists.
- The safety analysis of $\delta$-CORE. This part of work is summarized in a unpublished manuscript entitled "The Safety of $\delta$-CORE".

- The reliability and high-availability analysis of the system. A revised version of this work entitled "High-Availability in $\delta$-CORE: A Formal Derivation" appeared in Dedicated Systems Magazine, Q3, 2001[33].

Other generic features and properties of operating systems are also investigated:

- The testing of operating system and application programs. A set of programs $S$ is a test set of $\delta$-CORE if there exists a program $p$ in $S$ such that the execution of $p$ will result an error state for $\delta$-CORE.

- The compatibility analysis.

- The minimal models of operating systems. This part of work is based on the observation that a specific computation task can be achieved with a minimal resource and effort. A revised version of this part of work entitled “The Minimal Model of Operating Systems” appeared in ACM SIG Operating Systems Review, Volume 35, Number 3, July, 2001[34].

One lesson learn from this work is that since the formal model was established based on the denotational semantics of ToyC, It is totally functional. When it was finished, we have realized that there is one of the most important features not to be described. That is the interruption mechanism. We believed that we cannot talk an operating system without dealing with interrupts. Therefore, this language based formal model is not a complete one.

2.2 Formalization of A Partition-Based Operating System

Our second practice is the formalization of a partition-based operating system used for the aviation industry. This is a joint project with Air-Force of China and AVIC. The goal of project is to build a embedded real-time operating system ACoreOS which confirm to ARINC-653. This work is our first step of the project for fully understanding the ARINC-653 and the work was used as a guidance for the whole development process internally. The work was carried out following the same approach used for formalization of DeltaCORE. We simply modify ToyC by adding the partition mechanism into the language. The properties such as the spatial partition condition and temporal partition condition were analyzed. This work forms a technical report entitled “Partitioning Based Operating System: A Formal Model”. A summarized version of this technical report with the same title appeared in ACM Operating Systems Review, Volume 37, Number 3, July 2003[19].

2.3 Two Congruent Semantics of $\delta$-CORE

Our next work concentrates on the different equivalent semantics of DeltaCORE. In this work, we proposed two semantic models for the same version of DeltaCORE which is called DeltaCORE/S and proved that the two semantic models are congruent. This work forms a technical report entitled “Two Congruent Semantics of $\delta$-CORE/S” and is not published yet.

2.4 DeltaCORE and Monitor

Our another practise based on the language driven approach is an enhancement of ToyC with object-oriented and monitor mechanism called ToyMONITOR. ToyMONITOR has actually nothing to do with DeltaCORE itself. I was going to do it just by curiosity. As a result we obtained a technical report “The Formal Development of a Concurrent and Object-Oriented Programming Languages”. A byproduct of this work is a complete complexity analysis for system service calls of ToyMONITOR yielding a technical report “Toward to a More Predictable Real-Time Operating System Kernel”.

3
2.5 The Formal Specification of a Real-Time Operating System for Flight Control

Several years ago, we were involving on a flight control system on a single Intel X86 processor running the real-time operating system DeltaOS. However, after several successful testing flights, we found that there is a need to design and implementation of a real-time operating system specifically for flight control applications with the following features and requirements:

- All the tasks in the system are periodic tasks.
- The scheduling algorithm should be non-preemptive and static.
- There is no synchronization and communication mechanisms such as semaphore, message queue and event provided in the system. This means that all the tasks in the system are dependent tasks.
- It should reflect our programming practice and experience on flight control applications.

We have seen a number of domain-specific real-time operating systems in the market such as OSEK/VDX OS focusing on automotive control and ARINC-653 for avionics. However, there is neither a commercial real-time operating system design and implemented for flight control purposes nor a standard focus on this specific area. We have therefore seen the possibility to establish a fundamental framework toward to this direction. We are therefore decide to design a domain-specific real-time operating system for flight control applications which is named FliCOS.

FliCOS is designed specifically for embedded control applications. Consider a typical fly-by-wire flight control system, which consists of three types of interconnected components: sensors, CPUs for computing control laws, and actuators. For the cruise mode of the fly-by-wire flight control system, the sensors include:

- An inertial measurement unit (IMU), for measuring linear acceleration and angular velocity.
- A global positioning system (GPS), for measuring position.
- An air data measurement system, for measuring such quantities as air pressure.
- The pilot’s controls, such as the pilot’s stick.

Each sensor has its own timing properties: the IMU, for example, outputs its measurement 1,000 times per second; whereas the pilot’s stick outputs its measurement only 500 times per second.

Three separate control laws – for pitch, lateral, and throttle control – need to be computed.

The system has four actuators: two for the ailerons, one for the tailplane, and one for the rudder.

Beside the cruise mode, there are four additional modes: the take-off, landing, autopilot, and degraded modes. In each of these modes, additional sensing tasks, control laws, and actuating tasks need to be executed, as well as some of the cruise task removed. For example, in the take-off mode, the landing gear must be retracted. In the autopilot mode, the control system takes inputs from a supervisory flight planner, instead of from the pilot’s stick. In the degraded mode, some of the sensors or actuators have suffered damage; the control system compensates by not allowing maneuvers which are as aggressive as those permitted in the cruise mode.

Based on these observation, we were able to give a new design of RTOS for flight control which is completely different from the RTOSes we have built before. We have worked out a formal specification of the FliCOS. However, we were not able to get an implementation due to a number of reasons such as the acceptance of customers and the budget. This work forms a technical report entitled “The Formal Specification of a Real-Time Operating System for Flight Control in PowerEpsilon”.

2.6 Timer Management

This work presents a formal analysis of timer management mechanism underlying in embedded real-time operating systems. Many mathematical properties have been derived in the framework. This work is summarized in a unpublished manuscript entitled “Understanding Timer Management in RTOS”.
2.7 Virtual Machine and Operating System

This work is a personal project to provide a formal specification for a prototyping operating system kernel VMK which is a hypervisor-based system. This is again a reverse engineering work. Our chief engineer provided me the source code written in C language, I then rewrote them in terms of PowerEpsilon. The reason for this personal project is that I want to get the first experience on hypervisor technology.

Some formal analysis was done in this work, three properties have been verified:

- The separation kernel property.
- The security policy (information flow checking or capability checking).
- The correctness of semantic definitions of system service APIs against its formal specifications.

The work is shown in the technical report “A Formal Semantic Description of VMK - The Functional Specification”.

3 Formalization of Interrupt Handling

To give a formal model of interrupt handling mechanism we have to model an abstract machine in order to give a complete formal specification of embedded real-time operating systems. As it has been pointed out, this work was carried out with the following two reasons:

- It is impossible to talk about operating systems without mentioning the underlying machine.
- Interrupt management is a crucial part of any operating system and the specification of interrupt management can only be described in terms of a machine.

We start from an existing work - the Abstract Interrupt Machine AIM and then try to figure out a general framework for integrating interruption handling mechanisms into the formal models of operating systems.

3.1 Abstract Interrupt Machine - AIM

This work is inspired by the FLINT group at Yale University leaded by professor Zhong Shao on the advanced development of certified OS kernels. They have developed an abstract interrupt machine (AIM). Their work provides a foundation for reasoning about interrupt-based kernel programs and makes an important advance toward building fully certified operating system kernels and hypervisors[10, 9, 11].

This work is summarized in the technical report “Formal Design and Verification of An Elementary Real-Time Kernel in PowerEpsilon”.

3.2 An Operating System Machine: DeltaSVM

Our second work in this topic will be the development of an operating system machine for a hypervisor technology based virtual machine monitor (VMM) called SVMK (standing for Systematic Virtual Machine Kernel). We want to build a formal semantic model of operating system together with an interruption mechanism. In order to handle interruptions, we have to deal with machine instructions directly. Therefore we need to design an Operating System Machine.

The work was started by designing a simple machine and then extended by making all the operating system primitives which are non-interruptable as the machine instructions. So we may give another name to the work - “Operating System as an Extended Machine”.

This work is described in the technical report “An Operating System Machine”.

5
4 Security Policy in Operating Systems

Beginning from 2005 we have been intensively working on the partition and hypervisor base techniques for building new operating system products. The original goal for partition and hypervisor techniques are safety - the software fault isolation. However, after a couple of years of research and development, we found out that the partition and hypervisor techniques provide strong support for security. We then extended our work toward to this direction.

4.1 A Separation Kernel Formal Security Policy in PowerEpsilon

This work is a rework of security policy (GWV policy) for separation kernel by Greve, Wilding, Vanfleet[16] and Rushby[27] using the mathematical theorem proof development system PowerEpsilon. The aim of this work is going to fully understand the requirement specification of security policy for separation kernel. The core idea of GWV policy is the information flow control - an information can flow from one partition to another, if it is allowed to do so. This work is shown in the technical report “A Separation Kernel Formal Security Policy in PowerEpsilon”.

4.2 A Formal Model of Information Flow Control

This model is an enhancement of GWV policy dealing with the dynamic data structures in general purpose programming languages such as C[17]. In this model, the approach to modeling information in computing systems, while somewhat indirect, is one that focuses the location of information in the system, rather than the value of that information. In the model, the concept of information is defined relative to the system processing the information. We say things such as, “these inputs (the inputs located here) are classified as SECRET while these other inputs (located over there) are not.” Much of the focus of the investigation is in formalizing what is the meaning of “locations” and then connecting that model to the formal model of computation. This work is shown in another technical report “Information Security Modeling and Analysis in PowerEpsilon”.

5 Formal Model for Development Process of Embedded Real-Time Operating Systems

We were also pay a certain attention to the formal model for software development process of embedded real-time operating systems starting from the requirement specifications, function specifications, design specifications to implementation.

We have chosen FreeRTOS as the start point for investigation. The reason of choosing FreeRTOS for a starting point is that we think that we should start from a already existing small and open-source RTOS so that everything can be found and open for discussion. However, as someone pointed to me that FreeRTOS is rather complex comparing with other embedded real-time operating systems. In fact, it is true. Therefore we were facing a challenge for making the work done.

We then will be able to establish our own formal model of software development process for an existing project in CoreTek System to design and develop an unified kernel of RTOSes which is named DeltaUNITY.

5.1 Formal Model for Development Process of FreeRTOS

5.1.1 The Requirement Specifications of FreeRTOS

We start from an initial requirement analysis, then worked out the requirement specifications in a number of refinement steps. The whole work is shown in the technical report “The Formal Model of A Real-Time Operating System Kernel, Volume 1: The Requirement Specifications”.

6
5.1.2 The Functional Specifications of FreeRTOS

Following the requirement process, we then start the functional analysis. In these refinement process, the set-theoretic and predicative semantics of FreeRTOS is replaced by a completely functional semantics (denotational semantics). The work is shown in the technical report “The Formal Model of A Real-Time Operating System Kernel, Volume 2: The Functional Specifications”.

5.1.3 The Design Specifications of FreeRTOS

In the design analysis process, we will go more details inside of the system. We will talk about the underlying machine, the interruption mechanism and the code generation. The work is shown in the technical report “The Formal Model of A Real-Time Operating System Kernel, Volume 3: The Design Specifications”.

5.2 Formal Model for Development Process of DeltaUNITY

We were involving in a project to design and develop an unified kernel of RTOSes which is named DeltaUNITY. We have identified two alternative to reach the goal: a partial design kernel using generic programming and the hypervisor approach. Our engineering team has chosen the second scheme, I decided to try the both.

5.2.1 The Partial Design Kernel

The idea behind of the project is that we may separate a RTOS into two parts: the application-independent part and the application-dependent part.

- The application-independent part. This part isolates the interface with the hardware such as memory, registers, interrupts and I/O devices. It may also provides the configurable scheduling mechanism and the abstract interface to the application-dependent part.

- The application-oriented part provide the application programming interfaces of the specific RTOS to the programmers such as APEX, POSIX and OSEK/VDX, dedicated to avionics, general-purpose and automotive applications respectively.

More precisely we may divide an operating system into three layers.

- The nanokernel layer. A nanokernel delegates virtually all services - including even the most basic ones like interrupt controllers or the timer - to device drivers to make the kernel memory requirement even smaller than a traditional microkernel. This layer mainly focus on resource management and scheduling. The resources including tasks and domains. The task is the basic item for scheduling. The domain is the basic item for resource management where the domain is the low-level abstract concept for processes and partitions, that is, partitions and processes are built on domains.

- The layer of functional interface for kernel. To satisfy the requirement of application interfaces, this layer provides different standardized application programming interfaces including synchronization and mutual exclusion, communication in between tasks and domains, such as POSIX-like interfaces and APEX-like interfaces.

- The functional extension layer. This layer provides other functionalities beside of file system, databases, dynamic loading, networking protocols, device drivers, middle-ware and so on. This layer is designed according to the requirement from the application domains.

Whether it would be possible to come up with a “one size fits all” solution for RTOSes or not? This is the question we would like to answer. Since the traditional RTOS vendors did not succeed in capturing more than a half of the real-time market during the last 20 years, the rest part of market are consisting of home-grown solutions. We guess that the answer is no. This said that it should be possible to have a real-time framework which is extensible enough to allow people to build their real-time solution of choice.
over it. The system would simply provide a few canned interfaces in order to be usable out of the box, by people with common needs.

Simple design and straightforward code layers make the above possible. For instance, DSP-like applications do not necessarily need a full-fledged multi-tasking infrastructure; basic interrupt handlers with predictable preemption time are often enough here. On the other hand, you are not going to run a complete telecommunication software stack over interrupt service routines, so you need a complete RTOS core and rich utility libraries.

Based on these analysis, we will provide a partial design kernel framework which may work like a kernel generating system. The design process is conducted using PowerEpsilon starting from requirement specification, refining through functional specification, and finally obtaining a design specification.

This work is shown in our technical report “The Partial Design of An Unified Kernel for RTOSes”.

5.2.2 The Hypervisor Based DeltaUNITY

The Requirement Specification of DeltaUNITY

The informal requirement specification of DeltaUNITY given in natural language (in Chinese) in its initial version has the following drawbacks:

- It contains a lot of descriptions which are ambiguous in semantic description.
- Some of its definitions are incomplete.

The result of this work can be used as a supplementary material for the requirement specification of DeltaUNITY. We have therefore presented a rather complete requirement specification of DeltaUNITY. The completeness means that we are trying to give the definition as complete as possible, if some of the definitions are not possible to be described, we will give only the declaration and leave the details to the further refinement in the design process.

The work is shown in the technical report “The Formal Functional Specification of DeltaUNITY: An Industrial Software Engineering Practice”.

The Functional Specification of DeltaUNITY

This work is our continuous efforts on formal development of embedded real-time operating system kernel DeltaUNITY. Based on an formal requirement specification, we start from designing of an abstract DeltaUNITY machine, then provide a functional specification for the machine, and finally represent an extended GWV security policy together with the formal proof development in terms of PowerEpsilon[36, 37].

In our previous work, the functional specification of an abstract DeltaUNITY machine was given and the GWV security policy was briefly figured out in previous section. However, as we have pointed out that the information contained in the definition of GWV security policy is not enough for proving the GWV theorem which is our target of the project. In this work, the specification logic for GWV security policy will be extended for information flow analysis by introducing a data flow environment DFEnv defined as \[\text{Register} \rightarrow @\{\text{Set}, \text{Index}\}\] for tagging the information flow from labels of a VM to a register. The proof of GWV theorem will then be derived. The proof will be conducted by induction rules on MInstr and the sequence of MInstr - MInstrSeq with the well-foundness of SVMState.

The information flow analysis in this framework is simpler than what has been done in [17] where the information flow analysis was conducted based on C code level instead of being on an abstract machine code level. The complexity of information flow analysis on C code level comes from the pointers which may change dynamically from time to time.

The work is shown in the technical report “An Extension of GWV Security Policy in Abstract DeltaUNITY Machine”.
5.2.3 The Schedulability Analysis of DeltaUNITY

In this work, we present a schedulability analysis for the scheduling algorithm of DeltaUNITY. The work is shown in an unpublished manuscript “To Be Scheduled or Not To Be Scheduled, That is a Question! - The Scheduling Algorithm of DeltaUNITY”.

6 Formalization of BSPs and Device Drivers

6.1 Formal Models of ISRs

In this work, we have built several models of interrupt service routines for real-time operating systems in terms of PowerEpsilon. Beside of semantic specifications, the logical analysis of real-time properties are provided. This show that operating system design can be established with a very concrete mathematical background. The work is described in an unpublished manuscript entitled “Modelling of ISRs”.

6.2 The DEVice Interface Language DEVIL and Its Enhancement

In this work we present a formal semantic definition of a device interface language called DEVIL (DEVice Interface Language)[25, 26]. This work is intended for fully understanding semantics of DEVIL and will be used as a reference-guide in implementation project for δ-OS. A summary version of the work entitled “A Formal Semantic Definition of DEVIL” appeared in ACM SIGPLAN Notices, Volume 38, Number 4, April 2003[20]. A enhancement of DEVIL called DEVIL+ were developed for internal usage in CoreTek Systems[29].

6.3 A Semantic Model of Hardware/Software Co-Design for Device Driver Programming

A big problem on using DEVIL for driver development is that DEVIL can only be used for describing the syntax and static semantic attributes of a given device. The dynamic semantics of the devices such as the meaning of initialization, opening, closing, reading and writing of devices are totally missing in all the reference manuals. I have raised a question to my engineers and students: “who can write the first device driver when the hardware design completed?” The answer is “only the designers of device chip themselves!” I was shocked by the answer and finally realized that the answer is absolutely true. Our past practice on device driver programming was completely wrong in the sense that hardware and software were separately concerned. This will make the hardware/software integration very difficult. We believe that a device driver should not be considered as a software program only. Writing and testing device drivers is an interdisciplinary task that requires in-depth knowledge of both the hardware and software. Therefore, a device driver should be designed with the device itself as a whole thing. Not to be handled separately.

My solution to the problem is the following:

- A I/O device together with its driver is either an algorithm or a communication protocol implemented by both hardware and software. A hardware/software co-design forms a complete specification of the device.

- A programmer is greatly influenced by the language in which programs are written; there is an overwhelming tendency to prefer constructions that are simplest in that language. The best way for hardware/software co-design is to choose a VHDL oriented language for hardware description and a C oriented language for software coding.

I then start to workout a hardware/software model not for hardware design but for device driver programming so that when a I/O chip design is completed, its driver will also be generated. The idea behind of this work is that the input operation in a program of hardware (written in Verilog) may come from the output operation in a program of software (written in C), and vice versa.
The major contribution of this work is to provide a formal framework for a hardware/software co-design system model. We have presented a ToyV and ToyC co-design system, its syntax and semantics, and an analysis method for generating a decision tree guiding the device driver programming. This is our first step to attempt a general approach for generating device drivers from its Verilog design.

This work is shown in the technical report “A Semantic Model of Hardware/Software Co-Design for Device Driver Programming”.

7 Embedded Real-Time Operating Systems for Multi-Core Processors

This is a very recent joint project with AVIC initialized in 2015.

The requirement of this work is to establish the formal models of scheduling for multi-core architectures. Actually one of the biggest issues of the migration of a RTOS from the single-core to multi-core architectures is the scheduling. Since this is a purely research project, I have to identify a research goal by myself. What I have done is to investigate the issue in two aspects:

- To investigate the multi-core scheduling with a specific computer and operating system. This aspect of project will be the formal model a simple multi-core machine mcMARS.
- To investigate the multi-core scheduling without dealing with any specific computers and operating systems. This aspect of project is more like the formalization of some issues in Operational Research and Mathematical Programming.

7.1 The Operating System Machine mcMARS

7.1.1 The Core War Machine

As a starting point we want to design a simple machine as the basis for the investigation. What we have chosen is the Core War machine. Core War is a programming game created by D. G. Jones and A. K. Dewdney [5, 6, 7, 8] in which two or more battle programs (called “warriors”) compete for control of a virtual computer. These battle programs are written in an abstract assembly language called Redcode and run by a program called MARS (Memory Array Redcode Simulator).

At the beginning of a game, each battle program is loaded into memory at a random location, after which each program executes one instruction in turn. The object of the game is to cause the processes of opposing programs to terminate (which happens if they execute an invalid instruction), leaving the victorious program in sole possession of the machine.

7.1.2 Why Core War Machine

We want to start from a computing machine which has been shown to be insecure. The Core War machine is our choice, because it was designed for battles between programs. We will see how an insecure computer can be enhanced into a relatively secure one without changing its original instruction system.

7.1.3 From MARS to mMARS

We will design a Core War machine with the multi-core enhancement. The machine is called mcMARS standing for (multi-core Memory Array Redcode Simulator).

There will be four variations of the machines to be defined.

1. mcMARS 1 : Each partition is mapped into a unique core. So all partitions are running in parallel.
2. mcMARS 2 : Partitions are allocated into different cores statically. Each core is running a group of partition with statical scheduling.
3. **mcMARS 3**: Shared memory for multi-thread execution with dynamic scheduling where there is no thread synchronization mechanism.

4. **mcMARS 4**: Shared memory for multi-thread execution with dynamic scheduling and thread synchronization mechanism in built-in semaphores.

This is a codesign process intended to consider the functionality of hardware and operating system together to form a series of virtual machine architectures.

This work is showed in a technical report entitled “To Be Shared and To Be Separated - An Abstract Multi-Core Operating System Machine: Formal Specification and Analysis”.

### 7.2 Formalization of Multi-Core Scheduling in General

#### 7.2.1 Formalization of Periodic Multi-Core Scheduling

This work concerned with the formalization of theory of periodic scheduling for multi-core computer systems. Periodic scheduling considers the scheduling of operations that have to be periodically executed at a constant rate over time. Periodic scheduling problems arise in such diverse areas as production planning, real-time processing, vehicle scheduling, and personnel planning. The works was inspired by the paper [18, 4, 1, 12] on the topics of periodic multiprocessor scheduling, it was talking about multi-core scheduling without dealing with any specific computers and operating systems.

This work is showed in a technical report entitled “Formalization of Periodic Multi-Core Scheduling in PowerEpsilon - A Generic Framework”.

#### 7.2.2 Formalization of Multi-IMA Scheduling

This work is an extension of the previous work which concerned with a special case of periodic scheduling - the formalization of periodic scheduling for multi-IMA computer system.

The avionics industry has widely adopted the Integrated Modular Avionics (IMA) architecture, which supports the independent development of the various real-time avionics functions (having various safety-assurance levels) and enables them to run within partitions that are temporally and spatially isolated from one another. This partitioning mechanism has helped ease the certification process for avionics systems [28].

Over the past decade, microchip designers found it increasingly difficult to dissipate the waste heat that would be generated if more and more transistors were squeezed into each unit of area of silicon; this challenge has led to a leveling off of peak clock speeds. As a result, designers have assembled devices into multiple microprocessor cores as a way of achieving continued increases in computational power. Within the avionics industry, multi-core systems are attractive because they offer the potential for more easily achieving size, weight, and power (SWaP) requirements.

Migrating avionics systems from single-core systems to a multi-core system, however, is not a trivial task. The fundamental challenge to integrating an existing set of pre-certified single-core avionics IMA systems into a multi-IMA multi-core system is to ensure that the temporal and spatial isolation of the partitions will be maintained (and to achieve this assurance without incurring huge recertification costs).

Several issues arise in connection with meeting this fundamental challenge. One such issue is the difference in speed between a single-core processor and a multi-core processor; in general, one core of a multi-core system has lower clock speed than that of a single-core system due to temperature and power consumption constraints. Accordingly, a direct mapping, such as a one-to-one mapping, would not work if the speed difference is significant or existing partitions have irregular and inconsistent sizes.

Another issue is synchronization which arises in the migration to multi-core. Synchronization is required for exclusive transactions such as I/O, and helps achieve I/O virtualization. For example, some hardware devices such as graphic cards or storage units do not admit multiple accesses. In such cases, the IMA partition schedules should be synchronized by exclusive executions so that no data loss or corruption would occur.
This work is showed in a technical report entitled “Formalization of Multi-IMA Scheduling in PowerEpsilon”.

8 Development Tools: Compilers and Debuggers

I was wondering for a while whether to include this section or not. DeltaSystem does have a set of development tools named LambdaTOOL including compilers and debuggers. But we don’t make compilers and debuggers ourselves. We use actually the GCC and GDB. However, for a better understanding the technologies of compilers and debuggers. We do make a lot of efforts on formalization of compilers and debuggers.

8.1 Formalization of Semantics of Programming Languages and Compilers

Compiler generation based on constructive type theoretical setting begins with a specification, this is a representation of the relationship between the input language (source language) and output language (target language). Assume that we have a source language Toy.Prog and a target language TM.Prog – a T-Machine designed specially for language Toy, a specification should be a description of the expected semantic equivalence between these two languages. In general, the specification has the following form in terms of PowerEpsilon:

\[ \forall (x : \text{Toy.Prog}) \exists (y : \text{TM.Prog})@ (\text{Sem.Equiv}, x, y) \] (1)

Here, \( x \) denotes the input of desired program in source language and \( y \) denotes its output code in target language. The predicate \( @ (\text{Sem.Equiv}, x, y) \) describes the semantic relationship between the input \( x \) and the output \( y \). This specification is acting as a theorem, to derive a program from such a specification is equivalent to prove this theorem. The proof of this theorem must be constructive, in the sense that, in proving the existence of a satisfactory output \( y \), it must tell us how to find such an output. From this proof, a compiler to generate \( y \) from \( x \) can be extracted. What we seek is, in fact, a compiler \( f \) which returns \( y \) corresponding to each \( x \), so we may modify the specification to

\[ ?(f : \text{Toy.Prog} \to \text{TM.Prog})!(x : \text{Toy.Prog})@ (\text{Sem.Equiv}, x, @(f, x)) \] (2)

What will a proof of this formula consist of? It will be a pair \( \langle f, p \rangle \) with \( f : \text{Toy.Prog} \to \text{TM.Prog} \) and \( p \) a proof that for all \( x \) we have \( @ (\text{Sem.Equiv}, x, @(f, x)) \). This pair consists precisely of the function required together with a proof that has the property required of it.

A summarized paper of this work entitled “Denotational Semantics of Programming Languages and Compiler Generation in PowerEpsilon” appeared in ACM SIGPLAN Notices, Volume 36, Number 9, September 2001[31].

8.2 Formalization of Debuggers

In this work, we adopted an approach of using constructive type theory to derive a debugger of a given programming language from its denotational semantic definition. A summarized paper of this work entitled “Formal Specifications of Debuggers” appeared in ACM SIGPLAN Notices, Volume 36, Number 9, September 2001[32].

9 Conclusions

We have built a rather complete landscape of formal models for embedded RTOSes during the past 16 years. Most of the work was done in spare time by just personal enjoyment of the beauty and elegance of mathematics and computer science. As the result, each work yields a set of programs written in PowerEpsilon.
and a number of technical reports and manuscripts for documentation and summarization purposes, some of them has been revised and published, but most of them are not. I am planning to publish most of them in the near future if they are still valuable to the public.

The lessons learned from the long time work are the following:

- An operating system as a program has a very special attribute: it is a non-terminating program. Therefore it cannot be treated as a function mapping from an input to an output. This attribute will be impossible to make it as a pure mathematical object. The formal modeling of operating systems is more difficult than other subjects such as compilers and debuggers in computer science.

- The formal development of RTOS is a long process covering a number of refinement steps from requirement specifications, functional specifications, high-level and low-level design specifications, to finally coding. The transformation from requirement specifications to functional specifications and from functional specifications to design specifications are relatively easy. However, the transformation from design specifications to implementation in a programming language C will be very difficult. This step was usually done by code review - walk-through the hand-code against the design specifications. We are expecting that this step will be done automatically by a code generation tool.

- Operating systems are very complex software products. There is no common view on what is a correct operating system. Even we can formally give the definition for the correctness of operating system and prove that all the properties requested by customers are satisfied. The customers may still require to have the product to be tested. Our approach is based on the Curry-Howard interpretation of propositions as types and proofs as programs. However, the biggest difference is that programs can be executed on a machine. It will be a paradox that if we shown that the RTOS product is correct, it should be tested with zero defect. But for most of us, the result of zero defect means that the product has not been tested thoroughly. A compromised way may be to find a combination of validation and verification with proving and testing complemented with each other.

- Most of people in software industry has not accepted formal methods as the affective approaches for development of system software products such as operating systems and code generators in compilers yet. The formal methods were used as the complementary tools for documentation purpose only. The reasons are perhaps the following.

  1. The engineers are not well trained on formal methods. Most of them feel that writing a formal specification is more difficult than writing code directly. Therefore working on formalization is just a waste of time.
  2. There is an absent on good tools available in the area covering requirement analysis, functional specifications, design specifications and code generation. The hand coding is still the most affective way in the area.
  3. There is no international standard to force the usage of formal methods. DO-178C is an exception. But even in DO-178C the formal methods is still selective.

Any way there is still a long way to go.

The coming August 28 will be the 16th year anniversary of CoreTek Systems. This paper is dedicated to this special day. I want to send my grateful and congratulation to all of my employers and engineers in CoreTek Systems for their devotion, persistence and dreamful for a better, smaller, faster and securer RTOS kernel contributing to the information industry of world.

Happy birthday, CoreTek Systems!

References


