electronics

MDPI

Article

A Framework for Model and Verification of Safety-Critical Operating System Based on ARINC653

Wenjing Xu † and Dianfu Ma *,†

State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang Univerisity, Beijing 100191, China; xuwenjing@buaa.edu.cn

* Correspondence: dfma@buaa.edu.cn

† These authors contributed equally to this work.

Abstract: As the scale and complexity of safety-critical software continue to grow, it is necessary to ensure safety and reliability to avoid minor errors leading to catastrophic disasters. Meantime, the traditional method, such as testing and simulation alone is insufficient to ensure the correctness of systems. This leads to using formal methods to provide sufficient evidence for systems. However, design a high assurance safety-critical system by formal methods is challenging due to the complexity of operating systems. In addition, the traditional interactive theorem prover used in system verification requires hand-written proofs, which are more expensive. Therefore, the efforts of providing a standardized formal framework as well as safety proofs, are notable for the develop a safety-critical system. The purpose of this paper is to provide a safety framework to establish a highly reliable and safety-critical operating system based on the ARINC653 standard, a multilevel and standardized formal model. To verify the functional correctness of this model, we propose a context-based formal proof method for programs. To achieve this goal, we first model 57 core services of ARINC653 and define the high-level requirements as pre-and post-conditions. Then, we construct a set of specification statements a formal axiom system transformed into logical sentences, and the core service model is transformed into a logical sentence sequence to be proved. Finally, a context-based formal proof system for specification correctness is developed. We have verified the correctness of safety-critical operating system core services with this system. Experience shows that the verification system we developed can be achieved the functional correctness of a complete OS with a low implement burden, and that can simplify the difficulty of automated verification and increase the degree of automation of proof.

Keywords: formal verification; context-based; automated verification; ARINC653

Citation: Xu, W.; Ma, D. A Framework for Model and Verification of Safety-Critical Operating System Based on ARINC653. *Electronics* **2021**, *10*, 1934. https://doi.org/10.3390/electronics10161934

Academic Editor: George Angelos Papadopoulos

Received: 16 July 2021
Accepted: 9 August 2021
Published: 11 August 2021

1. Introduction

The role of safety and security in computer systems today is becoming more critical in the connected world. Due to the continuous increase in the scale and complexity of safety-critical software [1], how to verify the correctness of the program and improve the credibility of software has always been a matter of great concern in the computer academia. In recent years, with more and more formal methods being used in software verification and certification [2–4], software of general scale can be well verified, which can guarantee software more powerfully than traditional software testing methods. Existing formal methods of program correctness can make complete verification and certification at the program fragment level, but there are still few verifications at the system level for large-scale programs. As two relatively mature projects for the formal verification of microkernel operating systems: the Verisoft project [4] and the seL4 project, an actual operating system kernel was verified at the system level [5], and many test-based methods were successfully discovered an inevitable error. These two relatively successful formal verifications at the operating system level provide valuable experience for future operating system development and formal verification.

Formal model and verification of operating system (OS) is challenging work. The traditional method, such as testing and simulation alone is insufficient to ensure the correctness of or provide sufficient evidence for the certification of OS. This leads to using formal methods, which are mathematically-based techniques for the specification and verification of software systems, to ensure the correctness. To apply formal methods to the functional level of OS to verify its functional correctness, we first need to formalize their functional specifications. Formalization is difficult, error-prone, and non-automatic (for each OS component, it needs to be formalized from scratch), we need to understand the architecture and related knowledge of OS, and takes time to investigate when applying formal methods for formal modeling. In addition, the complexity of the function level (for example, number of components, concurrency, timing constraints, communication mechanism) often leads to formal modeling issues.

Because the relationship between the operating system (OS) kernel data structure is complex, the corresponding constraint conditions are also very complicated, making the constraint solver unable to determine it within the effective time range. Although automated proof tools can improve the verification efficiency of first-order logic specifications, due to the complexity of system software, no automated tool can reason and verify all system programs. For complex operating systems, how to use automated verification technology to formally verify the operating system is also a challenge [6]. Therefore, the functional correctness verification of complex system software is currently almost impossible to be fully automated, and human intervention is inevitable. So, how to reduce the cost of manual verification is an urgent problem to be solved in the construction of high-reliability system software.

In this paper, we propose a method of formal proof for program correctness based on context. It will provide a new idea for formal verification at the operating system level. We aim to formally prove the correctness of the safety-critical ARINC653 [7] (ScARINC653) operating system core services. For this reason, we transform the formal proof of the correctness of the ScARINC653 operating system core service program into formal evidence that these core service model (as a low-level requirement) is consistent with high-level requirements. We construct a set of formal axiom systems that transform program statements into logical statements, and transform the core service model into logical statement sequences; then we use the context-based formal proof method of program correctness to verify the correctness of ScARINC653 operating system core services. This method automatically takes the above-mentioned logical sentence sequence and high-level requirement pre-conditions as input passes through a series of context substitution operations and terminates context to match high-level requirement post-conditions.

To summarize, the primary contributions of this work are:

- We present a context-based proof framework, a new extensible architecture for building verified OS kernels. We introduced the operations between logical sentences into the formal proof process, and developed a context-based formal proof system for program correctness, to overcome the complexity of proof verification and the labor-intensive deficiencies to a certain extent.
- We build an abstract modeling language that used to model the 57 core services in the ARINC653 standard. We show how to impose the semantic transformation rules of the formal axiom system to transform the abstract language service model into a logical statement sequence.
- We show how the use of a context-based method at each layer allows us to apply our proof techniques for verifying formal specifications. Using our framework, we have successfully verified all core services of the ScARINC653 model. The verification process complies with DO-178C's [8] A-level software development and verification requirements.

The rest of this paper is organized as follows. Related work is shown in Section 2. The preliminaries of this paper are described in Section 3. Section 4 presents an overview of our methodology. Section 5 illustrates the formalizing proof systems. Besides, Section 6 is

devoted to 57 core services of ScARINC653 operating system for modeling and verification. Finally, Section 7 outlines the conclusion and future work.

2. Related Work

The seL4 project is based on the L4 microkernel and aims to complete a formalized, safe, high-performance operating system suitable for embedded systems [9]. The seL4 kernel contains more than 8700 lines of C code and more than 200,000 lines of Isabelle/HOL [10] proof script for formal specification definitions and certifications. The entire design and verification work takes a total of 20 person-years [11]. They have described the functional correctness proof of seL4 stretching from the abstract functional specification of the kernel to the binary file that builds by C code [11]. The functional correctness proof connects the C and binary implementations to the design and abstract specifications. The technique that they have used to express functional correctness formally is refinement, which is a well-known reduction of the method of forwarding simulation. They have presented the different refinement techniques used in the verification [12]. For the experience of formal verification of the seL4 microkernel, we knew that seL4 is currently not just the only general-purpose operating system kernel that is fully formally verified to this degree [13]. Zhao et al. [14] have presented the formalization and deductive verification of the ARINC 653 standard using EventB. The system functionality and all of 57 services specified in ARINC 653 Part 1 have been modeled in Event-B [15]. The above-mentioned work practices formal proof tools to manually write proof scripts to complete the verification of the system. The advantage of this verification approach is that the formal proof tools can generate evidence for the verification process of system software. However, its limitations are the low degree of automation and relatively high cost of verification.

The current research is mainly to use the program verifier to accept the standard annotated source code generation verification conditions and pass the verification conditions to the constraint solver to automatically solve the problem, such as Ironclad [16], Dafny [17], CertiKOS [18], Hyperkernel [19] and Serval [20]. The Ironclad project [16] is completely different from the previous operating system verification ideas. It treats the application, runtime library, and kernel as a whole, proving that the behavior of the entire system software on the hardware model conforms to the formal specification of the application layer. Ironclad uses an automated proof method based on SMT. It uses Dafny language with verification condition annotations [17] to implement runtime libraries and applications, and then Dafny automatically verifies the consistency of the program code and the verification conditions. Hyperkernel [19] is a method to design, implement and formally verify the functional correctness of the OS kernel. This method has a high degree of automatic proof, which reduces the burden of OS formal verification. It explores a push-button method to build a provably correct OS kernel. The data structure of Hyperkernel is designed to be effectively verified by the SMT solver. The kernel relies on arrays because the verifier can convert them into unexplained functions for effective reasoning. Serval [20] implements the reference counting encoding used in Hyperkernel. Serval is a framework for developing automatic verification procedures for system software. Serval provides an extensible infrastructure for creating verification programs by raising the interpreter below symbolic evaluation, and provides a systematic way to use symbolic analysis and optimization.

The above work is mainly to use the program verifier to accept the standardized and labeled source code generation verification conditions, and to pass the verification conditions to the constraint solver to automatically solve the problem. The advantage of this method is that the powerful SMT [21] solver greatly improves the automation of the verification process. The disadvantage is that it is difficult to complete all verifications of the correctness of complex system software functions. Because the relationship between the operating system kernel data structure is complex, the corresponding constraint conditions are also very complicated, making the constraint solver unable to determine it within a valid time range. Therefore, the functional correctness verification of complex system

software is currently almost impossible to be fully automated, and human intervention is inevitable.

3. Preliminaries

3.1. ARINC653

ARINC653 is an application program interface specification proposed for the integration of avionics system data [7]. The specification defines standard APIs and services between avionics operating systems and user programs. The ARINC653 specification divides core system services into six modules: partition management, process management, time management, inter-partition communication, intra-partition communication and health monitoring. Each module defines specific system services, a total of 57 core system services. The core module is divided into two layers, the core hardware layer and the core software layer. The core hardware layer includes a memory management unit, clock, interrupt controller, etc. The core software layer can be divided into two levels. The bottom level is highly related to the hardware. It provides an abstraction of hardware access and control, as well as the specific implementation of architecture, such as memory management, device drivers. The upper level is focusing on the realization of various mechanisms of the ARINC653 standard, such as partitioning mechanisms, scheduling, and communication mechanisms.

3.1.1. Partition Management

Partitioning is the core concept of ARINC653. Partition management service describes the mutual transformation of partition attributes, scheduling and partition mode. The partition includes data and code programs, which can be loaded into an independent address space in the core module. The partition mechanism allows the application in the module to have independent address space and time slices to execute through memory partition and time partition. In this way, the modules do not interfere with each other, which can effectively improve the safety of system.

ARINC653 provides two services for partition management, namely set and get partition mode. Partitions have no priority, and scheduling is strictly based on time. The partition scheduling algorithm is configured in advance through the configuration table, and the scheduling is continuously repeated in a main time frame. Each partition is assigned one or more fixed partition scheduling windows, and it is defined by its offset relative to main time frame and partition duration. The partition scheduler activates the corresponding partitions in turn according to partition window pre-configured in a main time frame, so that each partition has its own time part to access processor resources, thereby realizing time partitions. The resources of the partition operation, including channels, processes, queues, semaphores, events, etc., are specified when the system is created. The partition has four modes: cold start (COLD_START), warm start (WARM_START), idle (IDLE), and normal (NORMAL). After the initialization is completed, the partition enters the NORMAL mode, and the running program in the partition can change the partition mode through corresponding interface.

3.1.2. Process Management

The process management service describes the process attributes and the control execution, scheduling and status changes of the process under the partition. ARINC653 stipulates that in a partition, there can be multiple processes, and to meet the real-time requirements of system, processes in the same partition can be executed concurrently. The processes in the partition share address space of partition. At the same time, the process is the basic unit of concurrent execution of tasks in the partition. The process has four states: Dormant, Ready, Running, and Waiting. Among them, the process in the dormant state will not be scheduled, and the process is only in this state when it is just created or executed. A process in the Ready state can be scheduled to run, and it will become a running state after acquiring processor resources. A process in the Waiting state cannot be scheduled

either. Until the waiting event occurs, the process may be waiting for a certain resource, or it may be suspended while waiting. ARINC653 provides 14 process management services, including creating processes, obtaining process IDs, obtaining process status, suspending and resuming processes, and starting and stopping processes.

3.1.3. Time Management

Time management is particularly important for real-time operating systems. ARINC653 defines that the time of each partition in the module is unique and independent. All-time values or time capacities in a zone are defined or allocated based on zone time. ARINC653 defines four time management system services, namely time waiting, periodic waiting, acquisition time and replenishment time. The time waiting service causes the process requesting the service to suspend itself for a given time and regain the qualification to be scheduled after the waiting time has passed. The periodic process requests the period waiting service (PERIODIC_WAIT) at the end of the processing period to obtain a new time limit, which is calculated from the release point of the next cycle of the periodic process. The process obtains the system clock time by requesting the time service (GET_TIME). Replenishment time service (REPLENISH) enables the process to postpone its current deadline for a period of time-based on budget time.

3.1.4. Inter-Partition Communication

ARINC653 partition communication mechanisms include buffers, blackboards, semaphores, and events. The buffer and blackboard are used for general inter-process communication and synchronization, while semaphores and events are used for process synchronization. Both the buffer and the blackboard communicate indirectly, and messages are communicated through the buffer between processes. The difference between them is that the buffer allows messages to be queued, and the messages are stored in the message queue of the buffer in a "first in first out" (FIFO) order, and there is no information loss. The blackboard does not allow message queues, the messages in the blackboard will be cleared or overwritten by new messages. The semaphore is a counting semaphore, used to control access to resources by processes in the partition. Events are notified by setting the event to be waiting for the process of event occurrence, so as to perform synchronization work. An event usually consists of a binary state variable (up or down) and a collection of waiting processes. When a set condition occurs, the operating system sends a notification to the process waiting for the event.

3.1.5. Partition Communication

ARINC653 partition communication includes two or more partitions on a module to communicate. Partitions communicate with other partitions through messages, ports, and channels. The message is a continuous block of data, which is decomposed before being sent and reorganized before delivery at the destination. The channel defines the logical connection of communication between partitions and the transmission mode of messages from source to destination. There are two transmission modes of the channel: sampling mode and queue mode. Sampling mode stipulates that messages carry similar but updated data and message queues are not allowed. In contrast, the queue mode allows each new force of the message to contain different data. ARINC653 inter-zone communication includes two or more zones on a module to communicate. Partitions communicate with other partitions through messages, ports, and channels. The message is a continuous block of data, which is decomposed before being sent and reorganized before delivery at the destination. The channel defines the logical connection of communication between partitions and the transmission mode of messages from source to destination. There are two transmission modes of the channel: sampling mode and queue mode. Sampling mode stipulates that messages carry similar but updated data, and message queues are not allowed. The message is stored on the source port until it is sent out or is overwritten by a new message. The sampling mode indicates the maximum acceptable time for a

valid message through the refresh rate (Rrefresh Rate). In contrast, the queue mode allows each new force of the message to contain different data. Therefore, in the queue mode, messages can be queued at the source port until they are sent out, without message loss. After the message arrives at the destination port, it will be stored in the message queue of the destination port until it is received by the corresponding partition. ARINC653 defines 11 system services for inter-partition communication.

3.1.6. Health Monitoring

The ARINC653 standard has clear requirements for health monitoring. Health monitoring is used to respond to and report on hardware, application software, and operating system errors and failures. Health monitoring helps isolate errors and prevent failures from spreading. Because errors may occur at the module, partition, and process levels, health monitoring divides the error levels into module, partition, and process levels, and makes necessary repairs based on the fault level. Health monitoring maintains a log of faults and records the collected and detected system working status information for repair and use. Fault diagnosis can be carried out and reported through hardware, system configuration, or application.

4. Overview

In this article, we model and formalizes the core services of the ScARINC653 operating system. Our method establishes DO-178C conformant development and verification processes that are supported by formal methods. The overall architecture of operating system core service development and verification is shown in Figure 1.

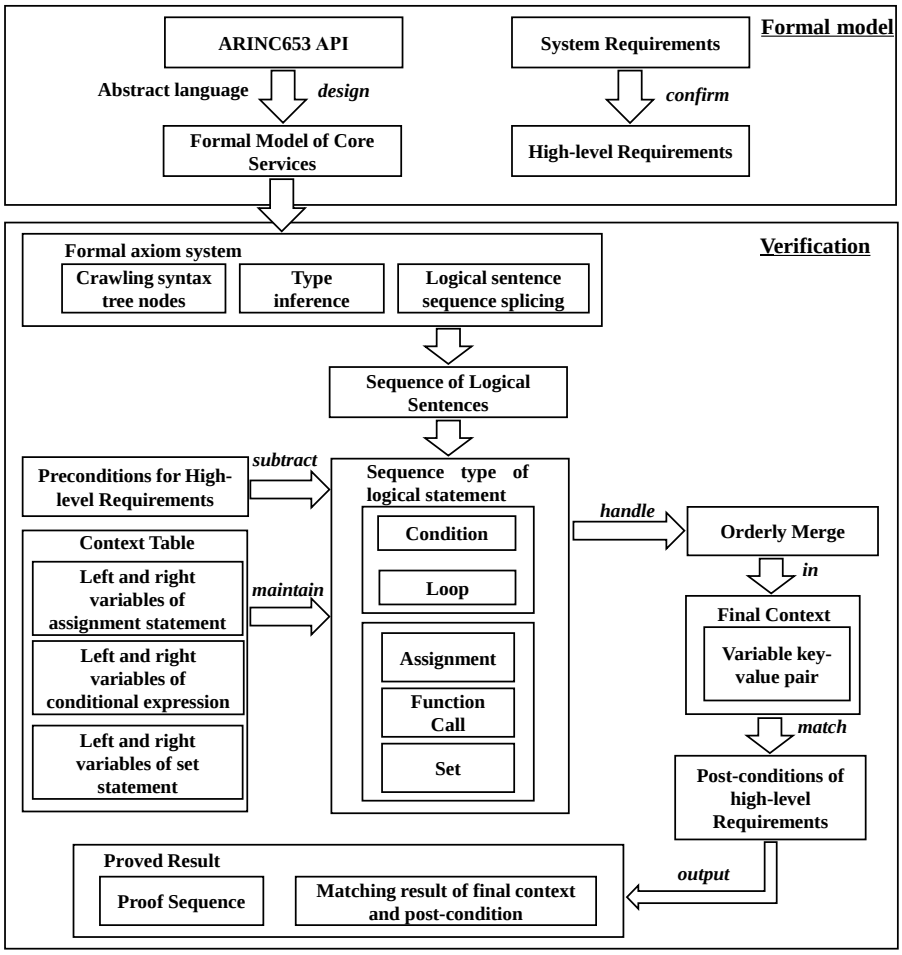

Figure 1. Model and verification architecture.

Our work is the formal development and verification of core services of a lightweight operating system, which includes formal modeling and formal verification. To achieve this goal, we designed the system services and basic functions of the ScARINC653 operating system established an abstract modeling language and used the abstract modeling language to formally model the core services of ScARINC653. We propose a formal method to prove the correctness of the model. We practice context to denote the context of program operation and apply abstract high-level requirements into functional models, which are described as pre-and post-conditions. Then, we exercise the high-level requirement pre-conditions as an initial context of the program to be proved and the formal verification of correctness is to verify the consistency of termination context for the post-conditions of high-level requirement. The formal modeling and verification process is as follows.

The operational context is introduced in Section 5, which is the basis of our model, and then presents the context-based formal proof method of program correctness. Then, based on the idea of formal proof, the proof strategies of basic sentences and collective sentences are given. We apply the context into the execution process of program and abstract the execution process of program. The execution process of the program is essentially the process of modifying the context variable. The operation of the context is to increase the variable name and update the constraint relationship of the corresponding variable value. Finally, some important algorithms in the formal proof method of program correctness based on context are analyzed in detail. One is the context update algorithm, which aims to start from the pre-order context and obtain a new context after the derivation and proof of the program statement. And the second is high-level requirements setting condition matching algorithm, which is used to match the final assignment of each variable in final context with the post-conditions of high-level requirements to obtain the result of the formal proof.

To formally prove the correctness of the ScARINC653 operating system core service modeling, we first use an abstract language to model 57 core services of ScARINC653 in accordance with the ARINC653 standard (see Section 6), and build a formal axiom system, and then design an automatic conversion algorithm to convert the form of the core service model into a series of logical sentences. Secondly, through analysis of system requirements, we determine high-level requirements and express them as functional pre- and post-conditions. To obtain the final context of each core service, we design automatic transformation rules, which start from the preconditions and the initial context, loop through the logical sentence sequence to process formal sentences and continuously update the context, and finally complete all sentences processing to get the final context. Finally, by verifying whether the final context matches the high-level requirements post-conditions, the high-level and low-level requirements consistency verification results are obtained, that is, the program correctness verification results.

The verification intention of this article is different from the current operating system formal verification project. We provide formal proof of functional correctness for the formalized model of abstract language middle layer, not the underlying code implementation. The advantage of this verification method is that errors can be found in the low-level requirement at the time, without waiting for the compilation and running of the underlying implementation code to find the errors. This is also the advantage of layering and the consistency of verification layers.

5. Methodology

5.1. Operational Context

In montague pragmatics [22,23], context is composed of tuples of function variables and context. For different contexts, the same variable can have different semantics. The operational context is a set of key-value pairs composed of the constraint relationship between the variable value and the corresponding variable value. The context ω can be expressed as:

$$\omega = \{\langle l_1, L_1\rangle, \langle l_2, L_2\rangle, \ldots, \langle l_n, L_n\rangle\}$$

Among them, l_1 to l_n represents several variable names, and L_1 to L_n represent the constraint relationship on the value of the corresponding variable. The uniqueness of the collection element is determined by the variable name.

The model of program execution can be expressed by the following formula:

$$\omega_0 \{ \Gamma \} \omega_n$$

It means that starting from a certain context $Context_n$, after the running of a program, the state of variables and data has changed and a new context $Context_{n+1}$ is formed. For the context *Context*, we abstract it as an operational context ω. Therefore, the process of running the entire program can be abstracted as starting from a certain context ω_0, the execution of program, the state of variables and data has changed, thus forming a new context ω_n.

5.2. Formal Proof Method of Context-Based for Program Correctness

In software engineering, the development of a program starts from the system requirements to determine its high-level requirements, to verify the correctness of low-level requirements, and then to system integration and testing. The high-level requirements describe what a program does and output the corresponding result in the case of certain input. Based on such high-level requirements, in the verification process of this article, first-order logic expressions can be used to express high-level requirements in the form of pre-and post-conditions, that is, under a certain pre-condition. After the program runs, the corresponding post-conditions are expected to be obtained set conditions. We assume it is correct for the high-level requirements of manual confirmation. Concerning how to prove the consistency of the low-level requirements and the high-level requirements and correctness of the modeling procedure, we make the following reasoning, where ω is the context concept proposed in the previous section, φ is the precondition, and χ is the postcondition, and Γ denotes the program to be proven.

$$\omega_0 \{\Gamma\} \omega_n \vdash \varphi \rightarrow \chi$$

The meaning of this formula is starting from the initial context ω_0 of the $\{\Gamma\}$ program, after a series of statements in the f program, the final context ω_n is obtained. This process can deduce the high-level requirement preconditions φ containing high-level requirements. The post-condition χ proves that the implementation of program meets high-level requirements, that is, low-level requirements are consistent with high-level requirements. Since there are multiple initial contexts in the Γ program, and the initial context is determined by high-level requirement preconditions, the high-level requirement preconditions are substituted into the context-based program execution process for subtraction, and the above deduction formula is modified, get the following formula:

$$\omega_0 \{\Gamma\} \omega_n \cup \{\varphi\} \vdash \chi$$

The meaning of deformed formula is Γ program takes the high-level requirement precondition φ as the constraint condition of its initial context ω_0, and the final context ω_n is obtained through the execution of the program. If the final context can deduce the post-conditions of high-level requirements, then it can be proved that the program fragment is consistent with the high-level requirements, so that the program is correct.

Above we have defined the core process of formal proof of program correctness, but it is impossible for us to directly prove a complete abstract language program. We need to decompose the program in order to prove and verify it sentence by sentence, and continuously update the value of the variable in the context and its constraint conditions, until all the sentences are executed, and the final context is obtained. Therefore, we can decompose the abstract language program Γ into a sequence of n sentences, namely

$$\Gamma = \Gamma_1, \Gamma_2, \ldots, \Gamma_n$$

The Γ_i here represents different types of statements, such as assignment statements, conditional statements, loop statements, etc. The sequence of Γ_i constitutes a complete Γ program. Each Γ_i sentence has the previous context ω_{i-1} as the initial context, and after the execution of Γ_i, the next context ω_i is obtained. According to this execution process, the running process of the entire Γ program can be further abstracted into the following formula:

$$\omega_0\{\Gamma_1\}\omega_1\{\Gamma_2\}\omega_2 \ldots \omega_{n-1}\{\Gamma_n\}\omega_n$$

Then the consistency verification of high-level requirements and low-level requirements can be reduced to the following formula:

$$\omega_0\{\Gamma_1\}\omega_1\{\Gamma_2\}\omega_2 \ldots \omega_{n-1}\{\Gamma_n\}\omega_n \vdash \varphi \to \chi$$

If it is proved one by one that the decomposed Γ_i program can deduce the step a, then the proof of the entire Γ program can be transformed into the following formula:

$\omega_0\{\Gamma_1\}\omega_1 \vdash \varphi_0 \to \chi_1$

$\omega_1\{\Gamma_2\}\omega_2 \vdash \varphi_1 \to \chi_2$

$\ldots$

$\omega_{n-1}\{\Gamma_n\}\omega_n \vdash \varphi_{n-1} \to \chi_n$

Substituting the high-level requirements precondition φ into the execution process of each step Γ_i program for subtraction, the formula is transformed as follows:

$$(\varphi_1 \to \chi_1) \cup (\varphi_2 \to \chi_2) \cup (\varphi_n \to \chi_n) \cup \{\varphi\} \vdash \chi$$

Perform the reduction process according to this formula to obtain the post-conditions of the low-level requirements. Compare the post-conditions of the low-level requirements with the post-conditions φ of the high-level requirements. If the post-conditions of the high-level requirements φ are a subset of the post-conditions of the low-level requirements, it means that the Γ program has realized the function of the high-level requirements. At the same time, the Γ program as a low-level requirement is consistent with the high-level requirement. What we mean here is that a Γ program will have multiple high-level requirements pre- and post-condition pairs. Only when the Γ program completes all high-level requirements pre- and post-condition pairs, can the Γ program be proved correct. Although the Γ program has multiple high-level requirements and preconditions, the number is far less than the number of test cases using the test method. Therefore, the formal proof method of the program correctness in this paper is effective and reduces the verification workload.

As mentioned above, we present how to formally prove the correctness of the program, and the process to verify the consistency of high-level and low-level requirements is completely provided.

5.3. Core Algorithm

In this section, we will discuss in detail core algorithms in the theorem derivation process: one is the context update algorithm, which aims to start from the pre-order context and obtain a new context through the derivation and proof of program sentences; the second is the high-level requirements post-condition matching algorithm, which is used to match the final assignment of each variable in the final context with the high-level requirements post-conditions to obtain the result of formal proof of program correctness.

5.3.1. Substitution Algorithm

The most important aspect of the formal proof method of program correctness based on context is the correct update of context. The rules of sentence derivation can only guarantee the correctness of the theorem proving process, and the result of sentence derivation is to modify the pre-order context to form a new context. From the initial context to the final context, the final result of each step of the deduction is a modification

of the context, so the context update algorithm is the most important link. The context update algorithm is divided into two processes, namely the substitution process (see Algorithm 1) and the merging process. These two processes will be explained in detail below. In the process of substitution, we make such a provision: the original context is called the pre-order context, and the context formed by the program sentence to be executed is called the subsequent context. In the pre-order context, there is such a relationship constraint for a variable $\langle l_1, L_1 = R_1 \rangle$, and in the subsequent context formed by the program statement to be executed, there is such a relationship constraint on the variable $\langle l_1, L_2 = R_2 \rangle$. Then the work to be done by the substitution operation is: Substitute the constraint relationship of l_1 in the pre-order context into the constraint relationship of l_1 in the subsequent context. The substitution operation will modify the subsequent context and form a new subsequent context.

Algorithm 1: Substitution Algorithm.

```
function substitute(RS1,RS2)
L = getL(RS1)
R = getR(RS2)
LRs = L ∩ R
if LRs ≠ null then
  for lr ∈ LRs do
    e = getExpr(RS1,lr)
    t = getType(e)
    if t == Direct then
      replace(RS2,lr,e)
    else if t == Condition then
          lr' = lr.generate()
          e' = e.clone()
          replace(e',lr, lr')
          insert(RS1,lr, lr')
          insert(RS1, lr', e')
          replace(RS2,lr, lr'
  end for
end if
end function
```

5.3.2. Merge Algorithm

The merging operation of the relationship set refers to the merging operation of $RS_2^{'}$ and RS_1 that have completed the substitution operation to form the final relationship set RS_t. For the part where there is no intersection between $RS_2^{'}$ and RS_1, insert it directly into RS_t. For the part where there is intersection, use direct coverage or conditional coverage according to different situations, and insert the merged intersection into RS_t. The merge operation algorithm is shown in Algorithm 2.

Algorithm 2: Merge Algorithm.

function merge (RS_1**,**RS_2**,** RS_t**)**
L_1 = getL(RS_1)
L_2 = getL(RS_2)
L = $L_1 \cap L_2$
L $\neq$ null
For $l_t \in L$
$l_t \in L_1 \wedge l_t \notin L_2$
insert(RS_t, l_t,getExpr(RS_1, l_t))
if $l_t \notin L_1 \wedge l_t \in L_2$ **then**
  insert(RS_t, l_t,getExpr(RS_2, l_t))
else if $l_t \in L_1 \wedge l_t \in L_2$ **then**
  e_2 = getExpr(RS_2, l_t)
  t_2 = getType(e_2)
if t_2 == Direct **then**
  insert(RS_t, l_t, e_2)
else if t_2 == Condition
    Cdt_2 = getPre(e_2)
    $Atom_2$ = getAtom(Cdt_2)
    CFlag = judgeComp($Atom_2$)
    if CFlag == All **then**
      insert(RS_t, l_t, e_2)
    else
      e_1 = getExpr(RS_1, l_t)
      $e_1^{'}$ = makeImp(getNot(Cdt_2),e_1)
      $e_2^{'}$ = makeAnd(e_2, $e_1^{'}$)
      insert(RS_t, l_t, $e_2^{'}$)
end function

5.3.3. High-Level Requirements Post-Condition Matching Algorithm

To confirm the correctness of the program, we use the idea of formal verification to design a matching algorithm (see Algorithm 3) for post-conditions of low-level requirements and high-level requirements to verify the consistency of two programs. If the low-level requirement post-conditions match the high-level post-conditions, the program is correct. The matching process of the algorithm is as follows: traverse the post-conditions of high-level requirement, and find the logical expressions of the corresponding low-level requirement post-conditions (PC) for all variables x. If the logical expression of the variable x of the post-condition of low-level requirement is consistent with the logical expression of the post-condition of high-level requirements, it can be matched to form a logical pair (PA). If corresponding logical expression cannot be found in the low-level requirement post-conditions, the logical pair output of the variable x is marked as empty. Only when all the high-level requirements post-conditions can find the corresponding logical expressions in the low-level demand post-conditions, can the correctness of the program and the consistency of the high- and low-level requirements be proved.

Algorithm 3: Match Algorithm.

```
function match(CX,PC,PA)
for x ∈ PC do
  ep = getExpr(PC,x)
  if x ∈ CX then
    ec = getExpr(CX,x)
    if ep= ec then
      insert(PA, ec, ep,x)
    else insert(PA, ec, ep,x)
  else insert(PA, -, ep,x)
end for
end function
```

6. Formal Verification of ScARINC653

6.1. Modeling Language and Logical Expression

In the traditional operating system verification process, the approach to operating system modeling is to take a subset of the underlying implementation language as the modeling language. However, the modeling language retains the complex data structure and operations of the underlying implementation language, so formal verification becomes very difficult. Because the use of a subset of the underlying implementation language as an abstract modeling language will have the above problems, we designed an abstract modeling language that can describe the most basic statements and operations of the core services of the operating system and it shields complex data structures and operations.

We build an abstract modeling language based on the grammar rules of the MATLAB software system, which can run in the MATLAB software system. We design five basic statements, which include assignment, function, loop, condition, and break, continue, and return statements. Compared with the traditional low-level implementation language, the control structure is greatly simplified, but it is enough to express the process of program operation. Some sentences and grammars of the abstract modeling language are shown in Table 1.

Table 1. Abstract Modeling Language Grammar.

Item	Grammar
assign-statement	<expr> ASSIGN <expr>
if-statement	IF <expr> (<statement>) (<elseif_state>) (<else_state>) END
while-statement	WHILE <expr> (<statement>) END
element-take-statement	id ASSIGN LEFTBRACKET element RIGHTBRACKET
function_call	id paralist
continue_statement	BREAK
break_statement	CONTINUE

Due to data structures such as arrays and linked lists, there are unsafe factors such as data overflow, array out-of-bounds, and complicated operations. The abstract modeling language abandons these unsafe data structures and replaces them with the most basic collection data structures and operations in logical operations. The traditional abstract modeling language does not have the concept and operation of a set. Therefore, this paper specially designs the corresponding set data structure and operation for the abstract modeling language and gives the corresponding set operation logic expression formal axiom system service.

6.2. Functional Requirements Analysis

Based on the ARINC653 avionics application software standard interface, this thesis uses formal methods to design lightweight operating system architecture, and conduct

formal modeling and verification of core services. The level of ScARINC653 modeling is concentrated on the core service layer of the system call. The reason for paying attention to this level is: the core service layer of system call provides services for ScARINC653. Modeling and verification of this level can ensure that ScARINC653 is provided with correct and reliable, Safe service; the core service of system call can best show the operating status of the operating system, and run with the software architecture (system scheduling mechanism, exception handling mechanism, system call function), and intuitively show the composition structure and operating mechanism of the operating system.

In the AINRC653 standard, each system service corresponds to its own functional system requirements. The system requirements are defined in the form of natural language and pseudo-code. We analyze the system requirements of each system service and establish high-level requirements based on pre/post conditions for each system service. This paper abstracts the system service function into a series of collections and operations on the collection in the high-level requirement stage, and the high-level requirement of each system service function corresponds to the pre- and post-conditions given in the form of logical propositions, which accurately describes The function of core system services eliminates ambiguity and ambiguity, and makes it possible to ensure that low-level requirements meet high-level requirements through formal proof.

Taking the process ID system service as an example, its high-level requirements are shown in Listing 1.

Listing 1. High-level Requirements.

```
void GET_PROCESS_ID (
\*in \* PROCESS_NAME_TYPE PROCESS_NAME,
\*out\* PROCESS_ID_TYPE *PROCESS_ID,
\*out\* RETURN_CODE_TYPE *RETURN_CODE ){};
(VALID (NAME)&&Pro.<NAME:PROCESS_NAME> : Process_Set ==> (*PROCESS_ID) == Pro.ID
&& (*RETURN_CODE) == NO_ERROR) &&
(~Pro.<NAME:PROCESS_NAME> : Process_Set ==> (*RETURN_CODE) == INVALID_CONFIG)
```

6.3. Core Service Modeling

We use an abstract modeling language to model 57 core services, which include partition management, process management, inter-partition communication, intra-partition communication, time management and health monitoring modules. For example, the model for get process id is illustrated in Listing 2. Process_Set represents the process set, used to record the properties and status of each process in the partition. In addition, the *ismember* function determines whether an element is in set.

Listing 2. Specification of GET_PROCESS_ID.

```
function [PROCESS_ID,RETURN_CODE]=GET_PROCESS_ID(PROCESS_NAME)
global RETURN_CODE_TYPE;
global Process_Set;

ProcessNameSet = {Process_Set{:,2}};
%APEX_INTEGER index;
[index,~] = ismember(PROCESS_NAME,ProcessNameSet);

if index==0
PROCESS_ID = -1;
RETURN_CODE = RETURN_CODE_TYPE.INVALID_CONFIG;
return;
end

PROCESS_ID = Process_Set{index,1};
RETURN_CODE = RETURN_CODE_TYPE.NO_ERROR;

return;
end
```

6.4. Semantic Transformation Rules

The purpose of designing sentence conversion rules is to convert abstract language programs into corresponding logical sentence sequences. In the formal proof of program correctness, all derivations are based on the first-order logic statement sequence. Therefore,

a set of semantic conversion rules must be specified first to convert the abstract language program into a first-order logic statement sequence. Whilst, constraints in the process of semantic conversion should also be specified. In the following rules, above the horizontal line is the given an abstract language program structure, and below the horizontal line is the first-order logic expression transformed into the program structure. The semantic transformation rule is shown in Figure 2.

$$\text{[SEQ]}\ \frac{\{S1; S2\}}{\{S1\}\{S2\}} \qquad \text{[ASSIGN]}\ \frac{x = e(R)}{x = e(R)} \qquad \text{[WHILE]}\ \frac{while(c)\ S}{(c \rightarrow S) \wedge (\neg c \rightarrow donothing)}$$

$$\text{[CONDITION]}\ \frac{if(c)\ S_1\ elseif(d)\ S_2\ else\ S_3}{((c \rightarrow S_1) \wedge ((\neg c \wedge d) \rightarrow S_2) \wedge ((\neg c \wedge \neg d) \rightarrow S_3))}$$

$$\text{[FUNCTION CALL]}\ \frac{func\_name(para_1 \ldots para_n)}{func\_name(para_1 \ldots para_n)} \qquad \text{[SET FETCH]}\ \frac{setA = setB(:, col)}{setA = setBcol}$$

$$\text{[SET DELETE]}\ \frac{set = setdiff(set, element)}{set = set - -element} \qquad \text{[SET FLAG]}\ \frac{flag = ismember(element, set)}{flag = (element : set)}$$

Figure 2. Semantic Transformation Rules.

Seq is a combined statement rule, which converts multiple sequentially executed statements into a continuous sequence of logical statements. The *Assign* statement will be directly converted into an expression with the equal sign (=) as the logical connector. The conditional statement uses the $\wedge$ logical connector to connect the implicit expressions of multiple conditions and results into one expression. Loop statements also use $\wedge$ logical connector, and only need to connect two implicit expressions into one expression. Function call statements are directly converted into logical statements of the same form. The set element-fetching statement is converted into an assignment logic statement with the equal sign (=) as the connector. The set insert element statement uses the –connector, which means that the element is deleted from the set, and then connected with the equal sign (=) as the assignment logic statement. The element belongs to the set statement using the belong (:) connector to indicate that the element belongs to the set, and then connect with the equal sign (=) as the assignment logic statement.

6.5. Logical Sentence Transformation

We use the semantic transformation rules of the formal axiom system in the previous section to transform the abstract language core service model of the lightweight operating system into a logical statement sequence. First, perform lexical analysis and grammatical analysis on the abstract language program that needs to be proved to obtain the syntax tree of the program. Then, read a sentence node from the syntax tree, and convert the syntax node into the corresponding logical sentence according to the conversion rules; finally, Traverse the entire syntax tree to get the logic statement sequence of the entire program. The algorithm for logical sentence transformation is defined in Algorithm 4.

Algorithm 4: Algorithm for Converting Program Statement to Logical Statement Sequence.t5

```
function parser(ParserContext, LogicalExpr)
  if ParserContext.assign_state ≠ null then
    LogicalExpr =visitAssign_state(ParserContext)
  else if ParserContext.if_state ≠ null then
      LogicalExpr = visitIf_state(ParserContext)
  else if ParserContext.function_call ≠ null then
      LogicalExpr = visitFunction_call(ParserContext)
  else if ParserContext.while_state ≠ null then
      LogicalExpr = visitWhile_state(ParserContext)
  else if ParserContext.element_take ≠ null then
      LogicalExpr = visitElement_take(ParserContext)
  else if ParserContext.element_ismember_set ≠ null then
      LogicalExpr = visitElement_ismember_set(ParserContext)
  else if ParserContext.element_delete_state ≠ null then
      LogicalExpr = visitElement_delete_state (ParserContext)
  else if ParserContext.element_insert_state ≠ null then
      LogicalExpr = visitElement_insert_state (ParserContext)
  end if
end function
```

As shown in Listing 3, we use this algorithm to convert *GET_PROCESS_ID* specification and obtain the transformed logical statement sequence.

Listing 3. Logical Sequence of GET_PROCESS_ID.

```
function[RETURN_CODE, PROCESS_ID]= GET_PROCESS_ID(ROCESS_NAME)

(ProcessName_Set = Process_Set{name})
(index = (PROCESS_NAME : ProcessName_Set))

((index$ == 0$)=>(RETURN_CODE = RETURN_CODE_TYPE.INVALID_CONFIG$))
((index$ == 0$)=>(PROCESS_ID = -1$))
((~ (index$ == 0$))=>(RETURN_CODE = RETURN_CODE_TYPE.NO_ERROR$))
((~ (index$ == 0$))=>(PROCESS_ID = Process_Set<@:id>$))

end
```

Thus, the formal modeling of core services of the ScARINC653 operating system is completed, and the model is transformed into a logical sentence sequence according to rules of formal axiom system.

6.6. Automatic Formal Verification of Functional Correctness

Based on the formal verification method of program correctness based on operating system modeling and formal axiom system and context, we design the automatic verification of ScARINC653 program correctness formal verification. The specific verification process (see Figure 1) is shown below.

First, we implement an abstract language lexical parser to read the input an abstract specification. Scan the input abstract language specification, analyze the lexical grammar according to the abstract language grammar paradigm, save the input abstract language specification with a certain data structure; convert its specification according to the semantic transformation rules of formal axiom system and form the corresponding logical sentence sequence, then use the corresponding data structure to represent the logical sentence sequence.

Secondly, scan the sequence of logical sentences to be proved to form the initial context. Then, from the sentence derivation rules and context update algorithm, the context is modified for each step of derivation. We practice a certain data structure to save the variables and constraint expressions in the context. Furthermore, the type of sentence will

be derived from the current context, then according to the derivation rules of the sentence, the evidence of the correctness of sentence is formed; at the same time, the follow-up context is formed after the derivation is performed, and the current context is used as input together to enter the context update algorithm, while a new context is formed. Next, traverse the high-level requirement pre-conditions, we use them as the pre-conditions of implication expression, and substitute them into the final context of specification after the inference is completed, and still apply the statement deduction rules to obtain the final assignment of variables in the low-level requirement post-condition.

Finally, traverse high-level requirements post-conditions, according to high-level requirements post-condition matching algorithm, compare post-conditions with the final assignments of variables in low-level requirements post-conditions, conserve the matching results with a certain data structure, and form the correctness of specification within the result of formal proof. For example, as shown in Listing 4, the post-condition of the high-level requirement is a proper subset of the final context. Therefore, *GET_PROCESS_ID* is decided to be correct.

Listing 4. Proof Result of GET_PROCESS_ID.

```
LOW-REQUIREMENT-POST : HIGH-REQUIREMENT-POST
-------------------- : --------------------
(PROCESS_ID = Process_Set<@:id>$) : (PROCESS_ID == Process_Set<@:id>)
(RETURN_CODE = RETURN_CODE_TYPE.NO_ERROR$) : (RETURN_CODE == RETURN_CODE_TYPE.NO_ERROR)
(ProcessName_Set = Process_Set{name}$) : null
(index = (PROCESS_NAME$ : Process_Set{name}$)) : null
```

7. Evaluation and Conclusions

7.1. Evaluation

We modeled and verified 57 core services of the ARINC653 operating system, and provided proof of the correctness of the program for the robustness of the parameters, the non-compliance of the preconditions and the normal operation of the program. The statistics for the effort and size of the specification and proofs are shown in Table 2.

Table 2. Specification and Proofs.

Item		Results of Work
Correctness Proof	Core Services	57
	Statement Lines	3200 lines
	Pre and Post Condition Pair	203
Tool	Grammar paradigm	183 lines
	Code Lines	7000 lines
	High-level Requirements	1228 lines

We describe the high-level requirements of 57 functions of ARINC653 operating system core services in the form of pre- and post-condition pairs, a total of 203 pairs, and use the grammar paradigm to model 57 core services with a total of 3200 sentences. We design verification tools to implement our framework. The main processing progress is: use the program analysis module to perform lexical and grammatical analysis of the program to be verified. After there are no grammatical errors, generate a syntax tree, and then use the logic sentence conversion module to convert the syntax tree into a logical sentence sequence; initial context generation The module scans the sequence of logical sentences and generates the initial context. Through the sentence derivation module and the context update module, the logical sentence is deduced and proved one by one to form the proof sentence and modify the context at the same time; use the high-level requirement preconditions to substitute the module to the final context. After subtraction, the final assignment of variables in the low-level requirement post-conditions is obtained; the high-level requirement post-matching module is used to compare the final assignment of the variables with post-conditions and the verification result is automatically generated.

We use our proposed method to verify the correctness of the modeling. The final statistical results are shown in Table 3.

Table 3. Problem Statistics.

Modules	Sub-Module	Number of Core Services	Problems
Partition	Partition	2	0
Process	Process	14	2
Partition Communication	Sampling	5	1
	Queue	6	2
Inter-partition	Buffer	5	1
	Blackboard	6	2
	Semaphore	5	3
	Event	6	3
Time	Time	4	1
Health Monitoring	Health	4	2
Total	10	57	17

We use the verification framework to take each pair of high-level requirements pre-conditions as the initial context, and the final context obtained after verification is consistent with the high-level requirements post-conditions. For example, to create a process service, we give multiple sets of high-level requirements before and after condition pairs to verify whether the core service function of the process is correct. When the high-level demand post-condition is a proper subset of the final context, the algorithm determines that the creation process under the high-level requirements is correct. Our verification results found that in the first round of modeling, there were 17 core service modeling procedures that could not achieve consistency with high-level requirements. According to statistics, the correctness of 57 core service models and the consistency of high-level and low-level requirements have been proved. It takes 200 milliseconds to verify each pair of pre-and post-conditions on average. We believe that there are implementation errors in them. Later, according to the high-level requirements and the low-level requirements description, the wrong program is corrected. After multiple rounds of modification and modeling, all errors found in the problems were corrected. This framework does save a lot of work, improves the degree of automated verification, and proves that the work of this article has practical value and significance.

7.2. Conclusions

In this paper, we aim to formally verify the correctness of system-level specifications that complies with DO-178C's A-level software verification requirements. We have presented a novel extensible framework for formulating verified ARINC653 core services that have not only an efficient modeling implementation but also machine-checkable contextual correctness proofs. We proposed a context-based formal verification method for specification correctness, which introduces the concept of context into specification verification. A context-based formal axiom system to specify how to convert program statements into logic statements is presented. To evaluate our approach, we develop a context-based formal verification system to automatically verify specification correctness, and we hope our practice can able to overcome to a certain extent the complexity of proof verification and proof burden of manual handwriting. The future work goal of this paper is to improve the method of formal proof based on the context of loop sentences and strive to accurately output the final context of loop sentences. Moreover, we also will learn Nickel [24] framework, a formulation of noninterference amenable to automated verification static noninterference, and we will design an automatic verification framework that formalizations and interface designs that are amenable to automated verification of dynamic noninterference [25] to verify security for ARINC653 core services.

Author Contributions: Methodology, D.M.; Project administration, D.M.; Writing—original draft, W.X.; Writing—review and editing, W.X. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Academic Excellence Foundation of BUAA for PhD Students.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DO178C	Software Considerations in Airborne Systems and Equipment Certification
ARINC653	The Avionics Application Software Standard Interface Set
OS	Operating System
ScARINC653	Safety-critical Operating System Based on ARINC653

References

1. Knight, J.C. Safety critical systems: Challenges and directions. In Proceedings of the 24th International Conference on Software Engineering, Orlando, FL, USA, 25–25 May 2002; pp. 547–550.
2. Woodcock, J.; Larsen, P.G.; Bicarregui, J.; Fitzgerald, J. Formal methods: Practice and experience. *ACM Comput. Surv. (CSUR)* **2009**, *41*, 1–36. [CrossRef]
3. Bowen, J.; Stavridou, V. Safety-critical systems, formal methods and standards. *Softw. Eng. J.* **1993**, *8*, 189–209. [CrossRef]
4. Bevier, W.R.; Hunt, W.A.; Moore, J.S.; Young, W.D. An approach to systems verification. *J. Autom. Reason.* **1989**, *5*, 411–428. [CrossRef]
5. Klein, G.; Derrin, P.; Elphinstone, K. Experience report: sel4: Formally verifying a high-performance microkernel. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, Edinburgh, UK, 31 August–2 September 2009; pp. 91–96.
6. Klein, G. Operating system verification—An overview. *Sadhana* **2009**, *34*, 27–69. [CrossRef]
7. ARINC. ARINC Specification 653: Avionics Application Software Standard Interface, Part 1, Required Services available from ARINC, SAE ITC, ARINC Industry Activities 16701 Melford Blvd, Suite 120 Bowie, MD 20715. Available online: https://www.aviation-ia.com/products/653p1-5-avionics-application-software-standard-interface-part-1-required-services (accessed on 23 December 2019).
8. *RTCA/DO-178C: Software Considerations in Airborne Systems and Equipment Certification*; Radio Technical Commission for Aeronautics: Washington, DC, USA, 2011.
9. Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; et al. seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; pp. 207–220.
10. Nipkow, T.; Paulson, L.C.; Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2002; Volume 2283.
11. Klein, G.; Andronick, J.; Elphinstone, K.; Murray, T.; Sewell, T.; Kolanski, R.; Heiser, G. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst. (TOCS)* **2014**, *32*, 1–70. [CrossRef]
12. Klein, G.; Sewell, T.; Winwood, S. Refinement in the formal verification of the seL4 microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 323–339.
13. Andronick, J.; Jeffery, R.; Klein, G.; Kolanski, R.; Staples, M.; Zhang, H.; Zhu, L. Large-scale formal verification in practice: A process perspective. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 1002–1011.
14. Zhao, Y.; Yang, Z.; Sanán, D.; Liu, Y. Event-based formalization of safety-critical operating system standards: An experience report on ARINC 653 using Event-B. In Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, MD, USA, 2–5 November 2015; pp. 281–292.
15. Abrial, J.R. *Modeling in Event-B: System and Software Engineering*; Cambridge University Press: Cambridge, UK, 2010.
16. Hawblitzel, C.; Howell, J.; Lorch, J.R.; Narayan, A.; Parno, B.; Zhang, D.; Zill, B. Ironclad apps: End-to-end security via automated full-system verification. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14), Broomfield, CO, USA, 6–8 October 2014; pp. 165–181.
17. Leino, K.R.M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 348–370.
18. Gu, R.; Shao, Z.; Chen, H.; Wu, X.N.; Kim, J.; Sjöberg, V.; Costanzo, D. CertiKOS: An Extensible Architecture for Building Certified Concurrent *OS* Kernels. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, 2–4 November 2016; pp. 653–669.

19. Nelson, L.; Sigurbjarnarson, H.; Zhang, K.; Johnson, D.; Bornholt, J.; Torlak, E.; Wang, X. Hyperkernel: Push-button verification of an OS kernel. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28 October 2017; pp. 252–269.
20. Nelson, L.; Bornholt, J.; Gu, R.; Baumann, A.; Torlak, E.; Wang, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, Huntsville, ON, Canada, 27–30 October 2019; pp. 225–242.
21. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
22. Dowty, D.R.; Wall, R.; Peters, S. *Introduction to Montague Semantics*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012; Volume 11.
23. Montague, R. Pragmatics and intensional logic. *Synthese*; Springer: Berlin/Heidelberg, Germany, 1970, Volume 22, pp. 68–94.
24. Sigurbjarnarson, H.; Nelson, L.; Castro-Karney, B.; Bornholt, J.; Torlak, E.; Wang, X. Nickel: A framework for design and verification of information flow control systems. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, 8–10 October 2018; pp. 287–305.
25. Eggert, S.; van der Meyden, R. Dynamic intransitive noninterference revisited. *Form. Asp. Comput.* **2017**, *29*, 1087–1120. [CrossRef]