End-to-End Automated Verification for OS Kernels

Jizheng Ding*, Xiaoran Zhu‡, Jian Guo†✉
*Soft/Hardware Co-design Engineering Research Center, East China Normal University
†National Trusted Embedded Software Engineering Technology Research Center, East China Normal University
‡Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Email: 51164500202@stu.ecnu.edu.cn, jguo@sei.ecnu.edu.cn

***Abstract*—Formal verification has been recognized as an essential part in improving the correctness and soundness of OS kernels. The verification of OS kernels faces many challenges. For example, the formal proof always comes with difficulties and non-trivial cost. Researchers often have to verify the OS kernel in high level programming languages and ignore low level languages. In this paper, we propose a framework for verifying the whole OS kernel with a low proof burden. We claim it as an end-to-end automated verification framework for the purposes of: (1) verifying the OS kernel that consists of assembler and C languages by formally modeling assembler programs in C level; (2) reasoning the verification with no manual proofs on the basis of SMT. We successfully apply the framework to automatic verifying a commercial operating system μC/OS-II in different hardware architecture, including all the 74 system calls and the core written in mixed-language (i.e., assembler and C). Our framework helps to catch several overflows and type mismatch vulnerabilities of μC/OS-II kernel.**

***Keywords*—Formal verification, Operating system, VCC, Mixed languages**

I. Introduction

As one of the most safety-critical components of software systems, an operating system kernel takes charge of processes management, memory protection, device drivers management, etc. It plays a decisive role in the performance and stability of the whole system. The presence of errors in operating system kernels may cause significant economic and human losses. How to ensure the safety of operating system kernels is an urgent problem. It is not easy to address this issue only using software testing. Formal methods have been proven as an effective auxiliary method to guarantee the correctness of a system.

There have been many notable efforts for adopting various formal methods on verifying operating systems. SeL4 [1] first demonstrates the feasibility of applying theorem prover HOL/Isabelle [2] to verify the correctness of the operating system kernel. Besides, Gu et al. use interactive theorem prover Coq to verify the correctness of a concurrent kernel in multi-core at assembly level [3]. Likewise, Xu et al. introduce a verification framework for verifying the implementation of an OS kernel by reasoning the conformance between the concrete implementation and abstract specifications in Coq [4]. To demonstrate the feasibility of the framework, they apply it to verifying the source code of μC/OS-II [5]. All of these impressive achievements rely on machine-checkable proof that requires manual interaction, and are expensive.

To reduce the ongoing effort for proving an OS kernel, researchers explore some approaches to automatic verification. Zhang et al. [6] and Zhu et al. [7] define an executable semantics for an OS kernel implementation language, with which they can automatically verify the correctness of OS applications and generate test cases for an OS kernel. Nelson et al. [8] propose an approach to verifying an OS kernel aiming at no manual proofs by compiling the high level language in the OS kernel into LLVM [9] and invoking Z3 to perform verification. Both of those two promising projects use idealistic sequential code written in a high level programming language. In practice, most of operating system implementations contain both assembler and C language. Assembler is used to access hardware devices that are not available in C or to improve the performance of critical components such as an interrupt.

While there are several verifiers for C based on SMT or model checking techniques, it is imperative to verify the assembler portion of an OS kernel as well. Several projects try to fill in this gap. [10] compiles C to assembler and conducts all verification at assembly level with Dafny [11]. Dafny is an automated verifier relying on SMT solver and is used for verifying the functional correctness of sequential programs. Stefan et al. [12] develop an automatic static analysis tool for verifying the Windows Hypervisor. Their work completely separates the C program from the assembler for modular verification and does not involve verification of the inline assembly. Baumann et al. [13] translate the assembler program to C level and verify PikeOS [14] automatically in VCC (Verifier for Concurrent C), which supports verifying concurrent C programs. Their paper introduces how to simulate the state transitions in C on the assumption of disabling the interrupt and avoiding context switches.

In this paper we propose a framework for verifying the whole operating system kernel including both C and assembly language. The framework makes it possible to validate the properties of assembler programs at the level of C code for verification. We bridge the gap between the two languages by formally modeling behaviors of assembler programs in an operating system kernel and simulating it in C. In this way, assembler programs can be integrated with C programs to achieve automatic verification. To demonstrate the feasibility of our framework, we apply it in verifying μC/OS-II on two different platforms, which is composed of more than 10,000 lines of C and around 100 lines of assembler. As a side effect, our work detects several vulnerabilities related to overflow and type mismatch in μC/OS-II kernel, which have been approved by the industry. To be more specific, our contributions are summarized as follows:

- An abstract model is built for the assembler that includes various states and the transition relations between states. Then the abstract model is integrated with the C implementation in a mixed language.
- The abstract model is implemented using a high-level language provided by VCC, and the gap between the implementation of the abstract model and the C code is bridged.
- Two implementation of μC/OS-II based on different platforms are verified. The verification result demonstrates the effectiveness of our method.

The rest of this paper is organised as follows. Section II introduces preliminaries. Section III introduces our overall framework. Section IV presents our abstract model of assembler programs. Section V introduces the implementation of the model in VCC. Section VI describes the verification of system calls and core part with mixed languages of μC/OS-II and analyzes the verification result. Finally, the last two sections evaluates and concludes this work respectively.

II. Background

In this section, we describe the knowledge is needed to verify μC/OS-II, including the μC/OS-II features and the fundamental principles and application of the verification tool.

A. *μC/OS-II kernel*

μC/OS-II [5] is a commercial real-time operating system, which has been widely used in many safety-critical systems, including avionics, medical devices and automotive. It is a priority-based, preemptive multitasking operating system kernel for microprocessors and microcontrollers. μC/OS-II manages up to 64 tasks with exclusive priorities. The four highest priority tasks and the four lowest priority tasks are reserved for its own use. This leaves 56 tasks for applications. The lower the priority value is, the higher the task priority is. μC/OS-II kernel provides a pool of different services and processing management, such as task scheduling, time management, communication mechanism and interrupt handling. Task scheduling is a preemptive priority-based scheduling strategy. Higher priority tasks can preempt lower priority tasks at any time. μC/OS-II is written in highly portable C, with target platform-specific code written in assembly language. C is to define system calls and scheduling. Assembler is to access registers and the stack related to hardware, or to improve the performance of critical part, such as interrupt handling and context switching. In order to ease porting to other processors, the main idea is to keep the platform-related code minimal. It is necessary to write or change the content of four CPU specific files for porting.

The platform of the commercial OS kernel to be verified is Freescale MCF5441x (abbreviated as MCF). MCF chip is composed of eight data registers (D0-D7), eight address registers (A0-A7), a 16-bit status register and some other components. Data registers are to store operands while address registers serve as software stack pointers, index registers, or base address registers. The hardware stack pointer of MCF is the eighth address register A7. The status register stores the processor status, the interrupt priority mask, and other control information. Interrupt priority mask is to set a bound for interrupt priority, i.e., interrupt can be handled only when its priority is greater than the interrupt priority mask [15].

B. *Hoare Logic*

Hoare logic [16] is a formal system with a set of logical rules for mathematically reasoning about computer programs. The logic was proposed in 1969 by Tony Hoare, inspired by Floyd ideas to define the program language semantics as a proof system.

The main idea of Hoare logic is to treat program specifications as contracts between the implementation of the program and its objects. The specifications are written in the pre-conditions and post-conditions. The pre-condition is a condition that the object must satisfy to ensure that the program can run properly. The post-condition is a condition that the object should satisfy after the program runs correctly. The typical notation of Hoare logic is so-called Hoare triple:

$$\{P\}S\{Q\}$$

where P is the pre-condition, Q is the post-condition, and S is the program statements. Pre-conditions and post-conditions are formulae in predicate logic. We say that the implementation program *partially correct* with regard to its specification if, S executes in a state initially satisfying P, and if the execution of S terminates, the post-condition Q is true.

C. *A Nutshell of VCC*

VCC (Verifying Concurrent C)[17] is a general-purpose automatic verifier for C. The first stage of VCC workflow translates the annotated C code into BoogiePL [18]. BoogiePL is an immediate verification language (IVL) for program analysis. The Boogie [19] tool takes BoogiePL programs as an input, then generates first-order logic (FOL) formulas which could be verified through an SMT solver Z3 [20]. Z3 may return two different result: (1) FOL formulae are verified. (2) Z3 returns a counter-example or timeout.

VCC regards everything as an object. Every object is independent of the other and has its own type and address. Objects can only be concrete or ghost [21]. The concrete object is the data defined in the program for execution, such as a C-defined variable, an array or a data structure and so on. The ghost object is defined by verification engineers for program reasoning. Transferring data directly between the ghost objects and concrete C objects is forbidden in VCC. Ghost statements are defined in the form of **_(ghost ...)**. And we use **_(def ...)** to define a ghost function, also called pure function, which has no side effects on the whole program.

The specification defined in VCC is called contract, which includes pre-conditions, denoted by **_(requires ...)** clause, and

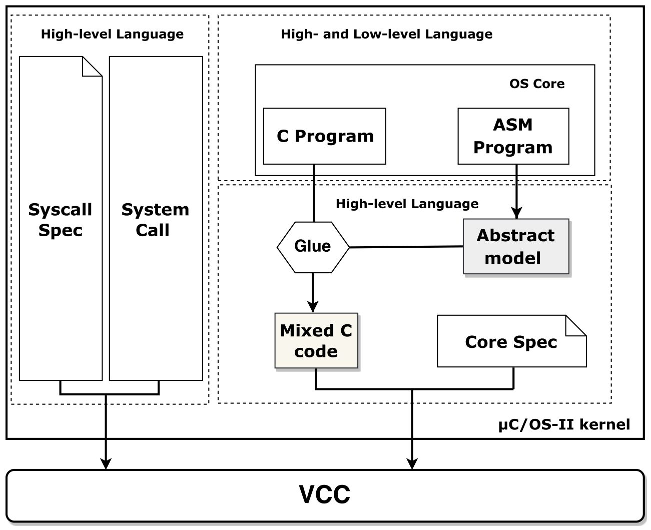

Fig. 1. Framework of Verification

post-conditions, denoted by **_(ensures ...)** clause. The pre-conditions and post-conditions specify the assertions that a function must hold in its pre-state and post-state. If a property is held in its execution, VCC provides us with **_(maintains ...)** clause to describe it.

Object invariants enforce that the object has only good properties. Such invariants can be captured by adding **_(invariant ...)** clause in the the data structure definition. An object is called **closed** if its all invariants hold. Otherwise it is called **open**. A closed object that is owned by the current thread is called **wrapped** object. There is a **_(wrap ...)** clause to change an object from open to closed. The **_(unwrap ...)** clause is to move the object from closed to open. More details about the usage of VCC can be found in [22].

III. Overview of Our Approach

In this section, we adopt μC/OS-II as a case to introduce the basic idea of our approach to verifying the operating system kernel. μC/OS-II is written in C and assembly language, for which we have proposed an automated verification framework to validate it.

Fig. 1 illustrates our method for verifying a whole OS kernel. The verification is divided into two steps. One is to verify system calls written in C language based on application programming interface specification. The other is to verify OS core written in the mixed-language, C and assembler languages.

We deploy VCC as the verifier in this work, VCC is an industrial-strength automated verification environment for C with the Z3 solver. It is designed for verifying concurrent programs, which is an important feature of operating systems. Particularly, VCC supports low-level C features, such as bit fields and weak type system. This allows us to simulate assembler instructions in VCC.

The structure of verification framework is shown in Fig. 1.

- *High-level language program verification* — the system calls of most OS kernels such as μC/OS-II are implemented by high-level languages, such as C language. In μC/OS-II, there are 74 system calls divided into 8 types. We take a system call as a module so that they can be verified separately and prevent state explosion from happening. Properties extracted from application programming interface specification are presented as Hoare Logic and data invariants. VCC invoking Z3 performs verifying all system calls properties automatically.
- *High- and low-level language program verification* — the OS core is written in both C and assembler to improve OS performance. Although VCC specification language is a property language, there is some syntax to describe communication with C program. We translate assembler programs to an abstract model with VCC specification language. In order to simulate the execution of assembler code in the abstract models. A glue between abstract model and C program is constructed. Properties about the OS core are extracted from the core specification and presented as Hoare Logic and data invariants. Properties including C and assembler programs are automatically verified with VCC.

The most intricate thing in verification is the validation of the mixed language. From the next section we start to introduce how we handle the mixed language.

IV. Abstract Model of Mixed Languages

As mentioned in Section II, μC/OS-II kernel is implemented in C and assembly language. In some cases, C functions may call some processor related functions which are written in assembler to access the CPU status and some registers. We regard those code segments where assembler functions are called by C statements as C with inline assembler. In some other cases, C functions could be called by assembler to improve the portability of the OS kernel. And these code segments are considered as assembler inline C code.

Some details of the approach about how to abstract the model for assembler programs are presented in this section. Furthermore, we also provide an example to illustrate how the approach works.

A. Abstract for Assembler

The first step of our approach to verifying mixed-language programs is to abstract assembler programs as a model, which could be represented as

$$\mathcal{T} : Prog_{C+ASM} \mapsto Prog_{C+Model}$$

where $Prog_{C+ASM}$ represents the C with inline assembler program. After transformation, the C with inline assembler program is turned into C with an abstract model ($Prog_{C+Model}$), which can be verified by C verifiers. The following part describes how we build the abstract model.

1) Syntax of MCF Assembly Language: Fig. 2 shows the syntax of partial MCF assembler instructions consisting of MOVE instruction, ORI instruction and LEA instruction. All

$$
\begin{array}{lll}
Instr & ::= Opcode\ src, des \mid \text{MOVE}.P\ \text{(sp)+}, des \\
 & \quad \mid \text{MOVE}.P\ src, des \mid \text{MOVE}.P\ src, -\text{(sp)} \\
Opcode & ::= \text{ORI} \mid \text{LEA} \\
P & ::= \text{B} \mid \text{W} \mid \text{L} \\
src & ::= \text{Dgpr} \mid \text{Agpr} \mid \text{[X]} \mid \text{SR} \mid (Val)\text{Agpr} \mid Val \\
des & ::= \text{Dgpr} \mid \text{Agpr} \mid \text{[X]} \mid \text{SR} \mid (Val)\text{Agpr} \\
Val & ::= Byte \mid Word \mid Int
\end{array}
$$

Fig. 2. The syntax of MCF assembler

assembler instructions consist an *opcode*, a source operand (*src*) and a destination operand (*des*).

The source operand can be an immediate value, a memory address, a data general-purpose register (Dgpr), an address general-purpose register (Agpr) with or without an offset value, or a status register (SR). The destination operand has a similar composition, except that it can not be an immediate value.

The MOVE instruction is to assign the value of *src* operand to the *des* operand. There are three forms of MOVE instructions, MOVE.B, MOVE.W and MOVE.L, operating on different types of operands. The MOVE instruction can also describe stack operations. Replacing the source operand with **(sp)+** indicates popping the stack, i.e., assigning the stack top value to the destination operand and then shifting down the stack pointer by one unit. Similarly, replacing destination operand with **-(sp)** indicates pushing a value onto the stack. In this case, the stack pointer should be shifted up by one unit before passing the value.

The ORI instruction performs bitwise or on the two operands, where the source operand should be an immediate value and the destination operand is a data general-purpose register. The LEA instruction loads the effective address into its destination. Note that the **sp** listed in Fig. 2 is an address register A7 which is used as a stack register pointer according to the specification of the MCF platform.

2) Auxiliary Function Definitions: All operations on the data in the assembler are stored in fixed-length registers, which are ultimately passed to the corresponding C program variables. However, the variables in the C program will have different types and different lengths, so there will be data conversion steps in the process of assembling the data to the C variable. Below we give the definition of the use of registers used in the assembly, as well as the rules for data conversion.

Definition 3.1 (**MCF data type definition**) The variable *x* could have one of the following five types.

$$
\begin{aligned}
\mathbb{N}_{32}(x) &\overset{def}{=} 0 \le x < 2^{32} \\
\mathbb{N}_{16}(x) &\overset{def}{=} 0 \le x < 2^{16} \\
\mathbb{N}_{8}(x) &\overset{def}{=} 0 \le x < 2^{8} \\
\mathbb{Z}_{32}(x) &\overset{def}{=} 2^{-31} \le x < 2^{31} \\
\mathbb{Z}_{8}(x) &\overset{def}{=} 2^{-7} \le x < 2^{7}
\end{aligned}
$$

Definition 3.2 (**Data conversion**) These two helpers are used to convert a natural number into an integer and convert an integer into a natural number respectively.

$$
\mathbb{N}_n\mathbf{2}\mathbb{Z}_z(x) \overset{def}{=} \begin{cases} x - 2^z, & 2^{z-1} \le x < 2^n \& n \ge z \\ x, & otherwise \end{cases}
$$

$$
\mathbb{Z}_z\mathbf{2}\mathbb{N}_n(x) \overset{def}{=} \begin{cases} x, & (0 \le x < 2^n \& z > n) \\ x, & (0 \le x < 2^{z-1} \& z \le n) \\ Invalid, & otherwise \end{cases}
$$

The value of n and z could be 8, 16 or 32. An *invalid* output represents overflow occurs.

With the definitions of data type and data conversion functions, we present some auxiliary functions to describe hardware processing. The hardware can only be accessed by assemblers and most of the assembly instructions operate on integers. Therefore, the data stored in registers can be described as:

$$
Type(GPR) \overset{def}{=} \mathbb{Z}_{32}
$$

We define predicates to demand the length of registers and its value to be an integer. GPR(A) stands for address general-purpose registers and GPR(D) represents data general-purpose registers in the predicates.

$$
\begin{aligned}
Type(GPR(A)) &\overset{def}{=} (|A| = 8) \wedge (\forall i < 8) \ : \ \mathbb{Z}_{32}(A[i]) \\
Type(GPR(D)) &\overset{def}{=} (|D| = 8) \wedge (\forall i < 8) \ : \ \mathbb{Z}_{32}(D[i])
\end{aligned}
$$

Definition 3.3 (**GPR read and write**) Reading and writing operations on a general-purpose register *r* at index *i* are defined by functions:

$$
GPR - read(r, i) \overset{def}{=} \begin{cases} r[i], & 0 \le i < 8 \\ 0, & otherwise \end{cases}
$$

$$
GPR - write(r, i, x) \overset{def}{=} \begin{cases} true, & 0 \le i < 8 \\ false, & otherwise \end{cases}
$$

B. Abstract Model for Assembler Program

This section describes how we build a model for assembler program and we also provide an example to show how it works.

1) Abstract Model: The execution of assembler code is similar to the operation of an automaton, which involves changes in program state, pointer changes, and the link between these changes.

Definition 3.4 (**Abstract model configuration**) Based on the syntax of assembler in Fig. 2 and the hardware components, we establish a transition system model to describe the assembler execution. Our model MCF_{ASM} is a triple which can be represented as:

$$
MCF_{ASM} \overset{def}{=} (S, sp, \delta)
$$

where S, sp, δ are respectively machine state, stack pointer and transition relation. In the rest of this part we introduce more details about this model.

Definition 3.5 (**Machine state**) Machine state S is a triple which can be formally defined as:

$$S \overset{def}{=} (dgpr :: \mathbb{Z}_{32}, agpr :: \mathbb{Z}_{32}, SR :: \mathbb{N}_{16})$$

The configuration of S is composed of three elements, *dgpr*, *agpr* and *SR*, representing data general-purpose registers, address general-purpose registers and the status register respectively.

Definition 3.6 (**Transition function**) Transition relation δ describes the whole assembler execution, which is defined as follows:

$$\delta \overset{def}{=} S \times sp \times Instr \rightarrow S \times sp$$

This function indicates how the state S and stack pointer sp are changed by executing an assembler instruction. The transition rules of assembly instructions in Fig. 2 are listed in the following:

TABLE I
TRANSITION RULES

Name	Expressions
Move	$(s, sp, MOVE.P\ src, des) \xrightarrow[\wedge des \neq sp]{src \neq sp} (s[des \Leftarrow src], sp)$
Read	$(s, sp, MOVE.P\ src, des) \xrightarrow[\wedge des \neq sp]{src = val(sp)} (s[des \Leftarrow (sp + val)], sp)$
Store	$(s, sp, MOVE.P\ src, des) \xrightarrow[\wedge des = val(sp)]{src \neq sp} (s, (sp + val) \Leftarrow src)$
Pop	$(s, sp, MOVE.P\ (sp)+, des) \xrightarrow{des \neq sp} (s[des \Leftarrow (sp)], sp \Leftarrow sp + size(P))$
Push	$(s, sp, MOVE.P\ src, -(sp)) \xrightarrow{src \neq sp} (s, (sp \Leftarrow sp - size(P); (sp) \Leftarrow src))$
ORI	$(s, sp, ORI\ src, des) \xrightarrow{src = val} (s[des \Leftarrow des \vert src], sp)$
LEA	$(s, sp, LEA\ src, des) \xrightarrow[\wedge des = sp]{src = val(sp)} (s, sp \Leftarrow sp + val)$

In the first one, the MOVE instruction passes the value of the src operand to the des operand. In case 2, *src* is a stack with an offset value and this instruction is used for reading data from a specified address of the stack. For example, there is a stack structure as shown in Fig. 3 (a). Before executing the instruction $MOVE.W\ 10(A7), D1$, the stack pointer points to the A7+0. After the execution of the instruction, the state s has changed for $D1 = 5$ but stack pointer has not moved. Similarly, storing to a specified address of the stack is depicted in case 3. Operations on popping and pushing the stack are shown in case 4 and 5, where size(P) represents the byte length of the data being handling. Case 6 depicts the transition rule of bitwise-or, where the source operand is an integer. The last one adjusts the position of the pointer and the operands of opcode *LEA* are the same stack register. With regard to the stack structure shown in Fig. 3 (a), the pointer shifts down by 4 bytes and points to the top of A7+4 after the execution of $LEA\ 4(A7), A7$. And then A7+4 turns into A7+0 as shown in (b). Note that the address register A7 serves as a hardware stack pointer.

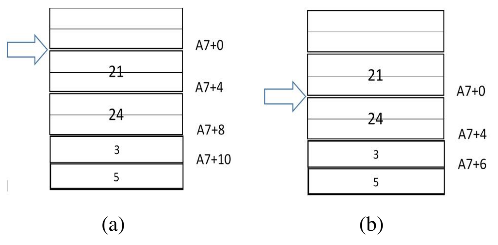

Fig. 3. Structure of stack

2) An Exemplary Abstracted Model: For a better understanding of our approach, we take an assembler implemented function *OS_CPU_SR_Restore()*, which is used to restore the interrupt control bits when the accessing to critical area resources ends, as an example to show how the transition system works.

```
        MOVE.L   D0,-(A7)       ; Save D0
        MOVE.W   10(A7),D0
        MOVE.W    D0,SR
        MOVE.L   (A7)+,D0       ; Restore D0
        RTS
```

The transition rules below demonstrate the execution process of function *OS_CPU_SR_Restore()*.

$$(s_0, sp_0, MOVE.L\ D0, -(A7)) \rightarrow$$
$$(s_0, sp_1 \Leftarrow sp_0 - 4, (sp_1) \Leftarrow D0)$$

$$(s_0, sp_1, MOVE.W\ 10(A7), D0) \rightarrow$$
$$(s_1 \Leftarrow s_0[D0 \Leftarrow (sp_1 + 10)], sp_1)$$

$$(s_1, sp_1, MOVE.W\ D0, SR) \rightarrow$$
$$(s_2 \Leftarrow s_1[SR \Leftarrow D0], sp_1)$$

$$(s_2, sp_1, MOVE.L\ (A7)+, D0) \rightarrow$$
$$(s_3 \Leftarrow s_2[D0 \Leftarrow (sp_1)], sp_2 \Leftarrow sp_1 + 4)$$

The first transition pushes data D0 into the stack after the stack pointer sp (A7) shifts up by 4 bytes. The second transition assigns the value of a specified address of stack to D0, which makes the state s changed. For the third one, the value of D0 is assigned to SR for restoring the interrupt control bits. In the last transition, it is a pop operation. The stack pointer sp moves down by 4 bytes and then the value pointed by sp is assigned to D0.

C. Modeling for Mixed Languages

The normal operation of the μC/OS-II kernel is inseparable from the cooperation between the C language code and the assembler code.

The collaborative executive process of the C language and assembler code in μC/OS-II is depicted in Fig. 4. Mixed languages in μC/OS-II are divided into two forms: C inline assembler and assembler inline C. The program that calls the assembly function in the C implementation is called C inline assembler. The assembler program that calls the C

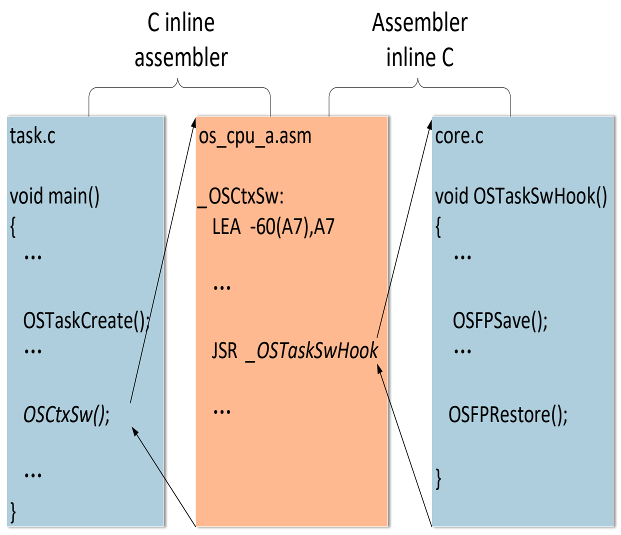

Fig. 4. Program calling sequence

function is called assembler inline C. The assembler program *OSCtxSw()* calls the C language program *OSTaskSwHook()* with an unconditional jump instruction JSR, which makes the program counter jump to a specified point. *OSTaskSwHook()* is used to extend the function of the task switching code in case the OS kernel code needs to use floating-point registers. *OSTaskSwHook()* does not contain any parameter and has no return value. In order to be able to access floating-point registers, *OSTaskSwHook()* needs to call two more assembler functions *OSFPSave()* and *OSFPRestore()*.

For modeling C inline assembler program, we first build an abstract model for the assembler program and directly replace the assembly call in the original C code, thus forming a mixture of C and an abstract model. Then using the tool to implement the abstract model to form a verifiable high-level language program. For modeling assembler inline C program, we directly insert the implementation of *OSTaskSwHook()* function into the higher assembler callers to replace the JSR instruction, which turns assembler program into C with inline assembler program. Then we can use the previous method for modeling C inline assembler program. The specific method for implementation of abstract model is introduced in the next section.

V. Implementation

In this section, we implement the abstract model mentioned in Section IV with VCC Ghost statement.

A. Syntax of VCC Ghost statements

VCC methodology heavily uses Ghost statements since they are transparent for C compiler and provide powerful capabilities for verification. So we consider implementing the abstract model with the help of Ghost statements. The syntax of Ghost statements, is defined in the Fig. 5:

$$\begin{array}{lll} \mathit{program} & ::= & \mathit{Ghost}\ \mathit{stmt} \\ \mathit{Ghost} & ::= & \text{ghost} \mid \text{def} \mid \text{typedef} \\ \mathit{stmt} & ::= & \text{X} = \mathit{Exp} \mid \text{Type } *\text{ptr} \mid \mathit{stmt}\ \mathit{stmt} \\ & & \mid \text{FunctionName(Type } \mathit{Id}\text{).Block} \\ & & \mid \text{struct varDeclaration}^{*}\ \mathit{Id} \\ \mathit{Exp} & ::= & \text{n} \mid \text{X} \mid e_0 + e_1 \mid e_0 - e_1 \mid \mathit{true} \mid \text{NOT } e \\ & & \mid e_0 | e_1 \mid e_0 << n \end{array}$$

Fig. 5. The syntax of VCC Ghost statements

Ghost statements are defined by the keyword **ghost**, **def** and **typedef**.

stmt contains one of an assignment statement, the function definition which is in form of $FunctionName(TypeId).Block$, the data structure definition which is in form of struct $\{varDeclaration*\}Id$ and a pointer definition.

Exp can be a constant *n*, an implementation variable X, a boolean expression, an arithmetic expression, a bitwise-or expression or shift operation etc.

B. Implementation of Abstract Model in VCC

To conduct verification of μC/OS-II kernel, we should implement the model abstracted in Section III in VCC. The State *S* is encoded by a global data structure, which contains three member variables representing data registers, address registers and a status register respectively. Since the status of CPU should be reset when restarting the OS, we need to initialize the SR (status register) with the statement $SR = SR.Init$ at the very beginning of the program.

```
_(typedef struct MCF_Hardware_struct{
  MCF_B32t D[8] ;
  MCF_B32t A[8] ;
  MCF_B16t SR ;
}MCF_c;)
  _(ghost MCF_B32t *SP)
```

where the MCF_B16t and MCF_B32t types are self-defined data, which correspond to 16-bit unsigned integer and 32-bit unsigned integer respectively. The stack pointer with the type of MCF_B32t is defined by a ghost statement in our VCC implementation.

The transition relation δ in the model is triggered by the execution of program statements and could cause state transitions. The following Ghost statements correspond to the transition relations defined in Table I.

$$\begin{array}{l} Case\ 1: des\ =\ src; \\ Case\ 2: des\ =\ *(SP+val); \\ Case\ 3: *(SP+val)\ =\ src; \\ Case\ 4: des\ = *SP,\ SP = \backslash old(SP)\ +\ 1; \\ Case\ 5: SP\ =\ \backslash old(SP)\ -\ 1,\ *SP\ =\ src; \\ Case\ 6: des\ =\ des\ |\ val; \\ Case\ 7: SP\ = \backslash old(SP)\ +\ val; \end{array}$$

Both of *src* and *des* above are members of the ghost structure MCF_c, and SP is a global stack pointer. Here we define the stack pointer SP to point to data with the length of 4 bytes, which implies that in some cases data conversion is needed. For instance, we use the transition relation δ to translate the instruction $MOVE.L\ 10(A7), D1$ into a ghost statement. Suppose we have defined a variable MCF with type of MCF_c, then the ghost statement is $MCF->D[1] = *(SP+10)$. In addition, the expression $\backslash old(e)$ in VCC represents the initial value of e on the function entry, which can be used for calculating the number of loops or reasoning the transition relations.

C. Glue between C and Abstract Model

The execution of an OS kernel needs interaction between variables, even between assembler code and C code variables. We use an abstract model MCF_{ASM} to simulate the execution of the assembler. However, direct interaction of C variables with variables of the MCF_{ASM} is prohibited in the verification of VCC. In order to solve this problem, we have to come up with a glue operation to bridge the gap between C and MCF_{ASM}. To accomplish this, we turn to the VCC specification statements.

```
_(def INT32U Assignment(MCF_B32t p)
    _(ensures \result == p)
...)
```

The post-condition $\backslash result == p$ on the entry of a function states that the function's return value should be p. In case of the function body is empty, VCC regards the post-condition as a tautology. Then we assign the return value of the *Assignment()* to a concrete object. By handling ghost objects in this way, we have successfully built a bridge between C implementation and the MCF_{ASM}.

VI. Verification of μC/OS-II Kernel

In this section, we introduce how to verify a μC/OS-II implementation with abstract hardware model. The verified modules include OS core and all system calls. The OS core is implemented by mixed languages.

A. Verification of System Calls

We use VCC to verify 74 system calls (6680 lines of C code), including task management, memory management, message queue, semaphore, mailbox, mutex, time management and flags. The verification work is based on the source code layer. Properties of program are formalized as VCC specification, and they are inserted in the corresponding point of source code. The verified system call modules are listed in Table II.

The column *Verified Modules* denotes the modules of system calls verified, such as the task management module, memory management module, message queue module, etc. Column *Syscall* shows the number of functions in different modules. Column *Specs* presents the lines of module specifications. Note that specifications are extracted from the requirement, and they are formalized as assertions, including pre-conditions, post-conditions and invariants. For the loops in the system calls, loop invariants are defined. They are maintained by an arbitrary iteration and hold on loop entry. A total of 653 lines of specifications of the system calls are proved. They are generally classified as the following 3 categories:

TABLE II
Workload of Syscall

Verified Modules	Syscall	Specs
Task management	12	127
Memory management	6	42
Message queue	10	103
Semaphore	7	63
Mailbox	7	59
Mutex	7	86
Time management	14	101
Flags	11	95
Total	74	653

- **Category 1** *Type Checking*
  It means that the type of arguments should satisfy the μC/OS-II requirements and there is no type mismatch. Type checking is an essential part of the whole system's correctness proving since type errors may lead to erroneous calculations. For example, in the assignments, such as $spoke = Match\%SIZE$, the arguments types on both sides of the assignment symbol should be consistent. We put an assertion *_(assert is_type_same(spoke,Match) && is_type_same(spoke,SIZE))* right before the assignment statement, where $is\_type\_same(par1, par2)$ is a self-defined pure function to determines whether two parameter types are the same. Similarly, we can add some pre-condition clauses to perform type checking on input arguments.
- **Category 2** *Invariants verification*
  For those properties that should be held during program execution, we specify them as invariants of the system call. For instance, the system call *OSMutexQuery(arg1, arg2)* is used to obtain information about a mutex. The argument $arg1$ is the mutex to be queried and $arg2$ stores the mutex associated information. In any cases, we need to make sure that the memory area for these two parameters are mutually exclusive. We specify two invariants, one with maintains clause, _$(maintains\ \backslash array_disjoint(arg1->Evtbl, tbl_sz, arg2->Evtbl, tbl_sz)$, located right at the entry of the function. And the other one with invariant clause, _$(invariant\ \backslash forall\ unsigned\ n < i, arg1->tbl[n] == arg2->tbl[n]))$, where i indicated the current number of cycles, is located at the entry of the loop statement inside the function body. The invariant clause requires the property holding for each loop.
- **Category 3** *Bound checking*
  Bound checking is used to ensure whether the value of a variable fits its type or the array index is in the bound of the array. We take the system call $OSTaskCreate(task, p\_arg, ptos, prio)$, which is defined to create a task with a specified priority $prio$

as an example. The priority range is between 4 and 59 for user. Priority *prio* is also used as an index to an array with size of 64. This property is specified as $prio \geq 4$ && $prio < 60$ presented in the pre-condition clause to assure that $OSTCBPrioTbl[prio]$ does not have an array out of bound, where $OSTCBPrioTbl$ is an array with size 64 to record the first address of the task control block with priority 0 to 63 tasks.

B. Verification of the Core

The core part of μC/OS-II kernel, which indicates interrupt handling and context switching, is implemented in mixed languages (i.e. C and assembler). To achieve the verification of the core part, we abstract the assembler program as MCF_{ASM} and simulated the model in VCC. We take the function *OSIntExit()*, which includes two forms of mixed languages mentioned in Section IV-C, as an example to introduce the verification for assembler programs in VCC.

OSIntExit() is used to notify the OS that interrupt service has finished executing, and it appears in pairs with *OSIntEnter()* in the interrupt services. *OSIntEnter()* is to increments and *OSIntExit()* decrements to the global variable $IntNesting$, which is the depth of interrupt nesting. After the last level of nested interrupts is completed, if there is a task is ready that is higher than the interrupted task priority, the task scheduling function is called. In this case, the interrupt is returned to the higher priority task instead of the interrupted task.

For the execution of core function *OSIntExit()*, external interrupts should be disabled for critical section access. The eighth to tenth bits of the status register are interrupt control bits. The status register is included in the abstract model and its implementation is the ghost object *MCF* with the type of structure *MCF_c*. An example of *OSIntExit()* is shown in Fig. 6.

Firstly, the function disables external interrupt handling and stores the status register to the data register D0 by invoking the pure function *SaveSR_DisableInt* (line 10), so that from here *OSIntExit()* begins to enter the critical section. Since members of abstract model *MCF* has to be modified by *SaveSR_DisableInt*, we add an *unwrap* clause in line 9. After external interrupts have been disabled, the invariant of *MCF* does not hold anymore. Before *MCF* is wrapped again (line 22), interrupt bits should be restored for holding the invariant (line 21).

Each execution of the *OSIntExit()* function decrements the value of $OSIntNesting$ by one. A task switch occurs when the interrupt nesting depth is 0 ($OSIntNesting == 0$), the scheduler is unlocked ($OSLockNesting == 0$), and the interrupted task is not the highest priority ready task (lines 12-15). Then the current task priority is set to be the highest runnable priority ($OSPrioCur = OSPrioHighRdy$), and values of all processor registers are restored from the stack of new task (lines 16-20).

```
1 void  OSIntExit (void)
2 _(writes OSIntNesting &&...)
3 _(requires OSIntNesting > 0)
4 _(maintains \arrays_disjoint(SP,16,MCF->D,
     8) && \wrapped(MCF))
5 _(ensures OSIntNesting == 0 && OSLockNesting
6   == 0 ==> OSCtxSwCtr == \old(OSCtxSwCtr) + 1
7    && OSPrioCur == OSPrioHighRdy)
8 {
9    _(unwrap MCF)
10   _(ghost SaveSR_DisableInt(MCF))
11   _(ghost cpu_sr = Assignment(MCF->D[0]))
12   ...//Dec. 1 of OSIntNesting
13   ...//If OSIntNesting && OSLockNesting==0
14   ...//Switch to highest priority task
15   OSTaskSwHook();
16   _(ghost MCF->D[0] = OSPrioHighRdy)
17   _(ghost OSPrioCur = Assignment(MCF->D[0]))
18   _(ghost SP = OSTCBHighRdy->OSTCBStkPtr)
19         _(ghost MCF->D[0] = *SP++)
20         ...// Restore regs from stack
21   ... //Restore SR
22   _(wrap MCF)
23 }
```

Fig. 6. An example for core verification

Finally, it is possible for VCC to verify the function *OSIntExit()* against its specifications listed in lines 2-7. Due to the space limit, we only list some representative specifications for understanding.

C. Verification Results

We verified μC/OS-II v2.83, which includes 13 C language files, two header files and one assembly language file with over 10,000 lines of code. The verification detects several bugs of the OS kernel, and they are presented in Table III.

The bugs can be classified into two categories:

- **Type mismatch**
  We found a case when data type conversion is erroneous. For example, bug 7 occurs in the assignment, $spoke = Match \,\%\, SIZE$ in the program. We found that the type on the left side of the assignment statement is INT16U while the right side is INT32U, which may cause catastrophic consequences for the whole system.
- **Data overflow**
  The reason why the occurrence of data transboundary phenomenon exists from problem 1 to 6 is insufficient program requirements, which is lacking of comprehensive data constraints description. For problem 4, in the scheduling function *OS_Sched()*, global variable $OSCtxCtr$ (context switching counter) is incremented by one, but no requirements describe to $OSCtxCtr$ when it is reset in the entire kernel code. Therefore, the program runs to a certain moment and this variable may overflow.

Our verification result has been approved by the developers. With the help of formal verification, the OS developed based on the kernel has passed the SIL4 certification.

TABLE III
LIST OF BUGS

No.	OS Kernel function	Property category	Bugs Description
1	OSTaskStkChk	3	Overflow.
2	OS_Sched	3	Overflow.
3	OSMemQuery	3	Overflow.
4	OSTmr_Free	3	Overflow.
5	OSTmrRemainGet	3	Overflow.
6	OSTmr_Task	3	Overflow.
7	OSTmr_Link	1	Type mismatch.

VII. EVALUATION

Scalability We have applied the verification framework proposed in this paper to verifying the implementation of μC/OS-II running on the MCF platform. There are more than 10,000 lines of kernel code (including C and assembler), while our verification consists 936 lines of apec and 205 line of abstract model. The proportion of our abstract model implementation code to assembly code is around 2:1, which is lower than our expectation.

Our verification framework is not limited to assembly language verification for a specific platform. It is general and can be applied to achieve verification on other platforms. With the purpose of demonstrating the feasibility and scalability of our verification framework, we make an attempt to apply it to the ARM Cortex-M3 (abbreviated as CM3 in the next) platform. Because the μC/OS-II kernel is highly portable, most of the code is hardware-independent, and only a small amount of assembler code is related to the hardware platform. As a sequence, when verifying the CM3 platform, we only need to abstract a new assembler model. For a detailed description of the hardware structure of CM3, please refer to [23]. Other hardware-independent parts can directly reuse the work of MCF. Table IV lists the comparison information for verifying the two platforms. Because the code of system calls is hardware-independent, so the reusability of system calls verification reaches 100%. The reusable verification condition of Core code reached 67.6% and the total reusability of the OS is 88.0%. According to Table IV, we believe our framework has capacity of verifying other μC/OS-II kernels based on any platform.

TABLE IV
VERIFICATION COMPARISON ON DIFFERENT PLATFORMS

Platform	System call	Core	Num. of bugs found
MCF	×	√	9
ARM CM3	×	×	11
Reusability	100%	67.6%	

Remark: Total reusability of the formal specification: 88.0%

Effectiveness As shown in Table IV, other than 9 bugs are detected in system calls. We have found two other priority inversion issues through the verification of the CM3 platform.

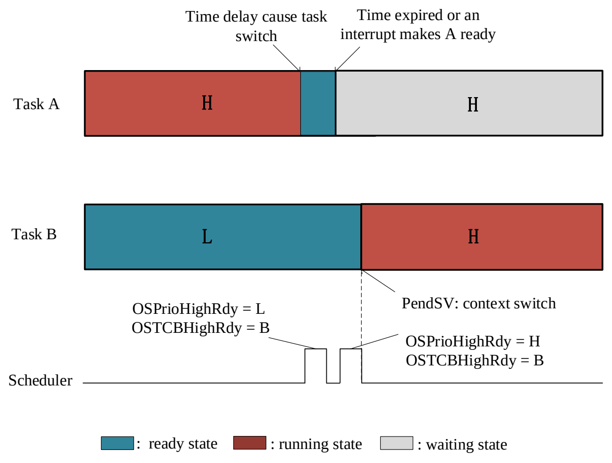

Fig. 7. Example of priority inversion

The reason for such issues are due to the hardware architecture of the CM3 platform.

In contrast, the reason for getting rid of priority inversion on the MCF platform is they implement context switching in different ways. Context switching on CM3 platform is implemented using PendSV, namely pendable request. PendSV is an interrupt driver with the lowest priority for system-level services, so it is controlled by the hardware interrupt control bits. Context switching on the MCF platform is implemented with trap instructions. The MCF trap instructions are not controlled by the interrupt control bits, and can be executed immediately even if the interrupt is disabled. Fig.7 outlines a case where priority inversion occurs on CM3. At the very beginning, high priority task A is in a running state with a priority of H, and low priority task B is in a ready state with a priority of L. Then task A is time delayed and immediately scheduler is invoked for the task scheduling. Before the execution of task scheduling, the interrupt control bit needs to be disabled, and the related information is updated so that task B enters the running state after the context switching. However, in the case of disabling an interrupt, context switching cannot be performed. If a new interrupt needs to activate task A during the disabled interrupt and scheduler is invoked again, updating the related information will cause confusion. As a result, PendSV is executed after the interrupt bit is enabled and there is no more other waiting interrupts, the running task is that task B with the priority of H.

Limitation The verification principle of VCC is to translate verification contracts into BoogiePL intermediate language, and finally into first-order predicate logic to be solved by Z3 solver. The Z3 solver is based on SMT with powerful and efficient reasoning and analysis capabilities. There are many formulas that can be verified using the inference capabilities of VCC, such as equalities, inequalities, addition, subtraction, multiplication by constants, and boolean connectives. But there are also some deficiencies, especially involving nonlinear arithmetic, bitvector reasoning. For example, VCC does not verify corresponding bits directly in bit-related operations. The following problem is encountered when verifying μC/OS-II on the MCF platform:

```
_(ghost MCF->SR = OS_INITIAL_SR)
...
_(ghost MCF->SR = MCF->SR | 0x0700)
    //Disabled interrupts
_(assert \forall int x,y;
 Bv_lemma(x|(y & MCF->SR),8) ==
(Bv_lemma(x,8)|Bv_lemma(y,8)))
```

Because the engine of VCC cannot calculate the bit operation during the verification process, here we use a function defined by a Bv_lemma macro to implement bit verification, where Bv_lemma can be used for bit-dependent verification.

Since our specifications are extracted from the requirements provided by the operating system kernel development team, the limitations of the requirements may lead to our specifications not complete. However, we believe our framework can be applied to a more complex and larger mixed program.

VIII. Conclusion

In this paper, we present an end-to-end automated verification framework. Our framework supports certifying low-level programming at a high abstraction level without sacrificing precise control over hardware resources. We demonstrate the effectiveness of our framework by applying it in verifying an industrial OS kernel (μC/OS-II) on two different platforms. Our experiment shows that it is feasible to abstract the mixed code containing the assembler to the high level to verify the functional correctness of the OS kernel. Meanwhile, we have detected some potential deficiencies on different platforms. With the help of our verification framework, the reliability of operating system development has been greatly improved. We believe our framework offers a promising direction for verifying the systems implemented by mixed-language and it may be extended to other more complicated mixed-language scenarios.

Current work involves functional verification for scheduling mechanism, interrupt handling, resource management of the OS kernel. To achieve full verification of μC/OS-II, memory protection mechanism and information flow security are two indispensable parts. One limitation in our work is conducted at a controlled interrupt situation. Besides, since μC/OS-II is a real-time operating system, the time property analysis is another topic we would like to pursue.

Acknowledgement

This work was supported by China HGJ Project (No. 2017ZX01038102-002), National Natural Science Foundation of China (61532019, 61472140).

References

[1] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, 2010.

[2] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[3] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent OS kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016.*, 2016, pp. 653–669.

[4] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li, "A practical verification framework for preemptive OS kernels," in *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, 2016, pp. 59–79.

[5] J. J. Labrosse, "Microc/os-ii," *R & D Books*, vol. 9, 1998.

[6] M. Zhang, Y. Choi, and K. Ogata, "A formal semantics of the OSEK/VDX standard in $\mathbb{K}$ framework and its applications," in *Rewriting Logic and Its Applications - 10th International Workshop, WRLA*, 2014, pp. 280–296.

[7] X. Zhu, M. Zhang, J. Guo, X. Li, H. Zhu, and J. He, "Toward a unified executable formal automobile os kernel and its applications," *IEEE Transactions on Reliability*, pp. 1–17, 2018.

[8] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an OS kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 252–269.

[9] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[10] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14.*, 2014, pp. 165–181.

[11] K. R. M. Leino, "Accessible software verification with dafny," *IEEE Software*, vol. 34, no. 6, pp. 94–97, 2017.

[12] S. Maus, M. Moskal, and W. Schulte, "Vx86: x86 assembler simulated in c powered by automated theorem proving," in *International Conference on Algebraic Methodology and Software Technology*. Springer, 2008, pp. 284–298.

[13] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal verification of a microkernel used in dependable software systems," in *Computer Safety, Reliability, and Security, 28th International Conference, SAFECOMP. Proceedings*, 2009, pp. 187–200.

[14] R. Kaiser and S. Wagner, "Evolution of the pikeos microkernel," in *First International Workshop on Microkernels for Embedded Systems*, 2007, p. 50.

[15] F. Semiconductor, "Mcf5441x reference manual," 2012.

[16] M. Gordon, "Background reading on hoare logic," 2012.

[17] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics, 22nd International Conference, 2009. Proceedings*, 2009, pp. 23–42.

[18] R. Deline and R. Leino, "Boogiepl: A typed procedural language for checking object-oriented programs," *Microsoft Research*, 2005.

[19] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 364–387.

[20] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS. Proceedings*, 2008, pp. 337–340.

[21] D. Škorvaga, "Analysis of a file system using the verifying c compiler," pp. 12–421, 2015.

[22] M. Moskal, W. Schulte, E. Cohen, M. A. Hillebrand, and S. Tobies, "Verifying c programs: a vcc tutorial," *MSR Redmond, EMIC Aachen*, 2012.

[23] J. Yiu, *The definitive guide to the ARM Cortex-M3*. Newnes, 2009.