Towards an automatic verifier for the OS interrupt mechanism

[*]Note: Sub-titles are not captured in Xplore and should not be used

Anonymous

***Abstract*—Formal methods have been recognized as complementary means to software testing to improve the reliability of systems. Formally analyzing a system or a program requires formal semantics of its implemented language or specification to avoid ambiguous interpretations. To our best knowledge, most of the semantic analysis tools face on monolingual implementations, while some safety-critical systems are implemented in more than one languages (mixed language). For example, most of operating system kernels are implemented in assembler and high level language (e.g., C language). In this paper, we propose an approach to automatically verifying systems implemented in mixed language. We define operational semantics for the high level language and assembler respectively, and then integrate them together. The integrated semantics is implemented in a rewrite-based executable semantic framework named $\mathbb{K}$, with which we verify a demo loop program implemented by mixed language. Furthermore, we demonstrate the effectiveness of our semantics by adopting it on verifying the schedule mechanism of an industrial automobile operating system kernel.**

***Index Terms*—formal semantics, verification, operating system, $\mathbb{K}$ framework**

I. Introduction

As a complementary approach to testing, formal methods have been proved to be an efficient way to guarantee the correctness and soundness of software system. Formal methods has the capability to conduct an exhaustive exploration of the software behaviors, which is necessary for ensuring correctness of safety-critical systems such as operating systems and an exhaustive systems. Formal analysis of systems or programs should be based on the formal semantics of the implemented language rather than on the informal specification because informal specification may cause ambiguities and mislead the developers.

There are many challenges on formally verifying operating systems. For example, operating systems are usually implemented in a mix of assembly and high-level language like C. To our best knowledge, there is no formal tools which could verify a system implemented in a mix language. A traditional approach to verifying such a system is to abstracting the assembly part to a high level. However, it is difficult to ensure the soundness of the abstraction. Besides, developers need to abstract a model for every system, which brings some extra effort in formal verification. Another way is to translate high level language to assembly, where a fully verified compiler (or translator) is necessary.

Recently, some researchers are paying attentions to defining semantics for mixed language to verify a system implemented in more than one languages. For instance, Schmaltz et. al [1] integrate the semantics of C and assembler to verify operating systems and hypervisors. Pentchev extend the work [2] with the semantics of interrupt processors. However, their semantics are defined by pen-and paper, and it is not easy to check the soundness of it. Besides, the analysis of operating system based on their semantics is by hand, which is non-trivial. Thus a checkable semantics with an automated toolkit for system analysis is expected.

$\mathbb{K}$ [3] is a rewrite-based executable semantic framework for program or system analysis. The semantics defined in $\mathbb{K}$ is testable thanks to its executability. Besides, $\mathbb{K}$ provides a set of program analysis tools, such as model checker [4], deductive program verifier [5] and a cross language program equivalence checker [6]. $\mathbb{K}$ has been successfully applied in many industrial languages, such as C [7], java [8] and javasript [9]. In particular, C semantics in $\mathbb{K}$ passed 99.2% of the test cases for GCC [10], and has been commercialized by companies to verify the correctness and safety of C programs. Furthermore, $\mathbb{K}$ is being deployed to design a new language for secure systems, such as the blockchain system.

With the aim of automatically formally analyzing systems implemented in mixed language, we adopt $\mathbb{K}$ as our semantic analysis tool in this paper. Firstly, we extend the IMP language by adding the function call and function return statements. Then we define formal semantics for IMP with function call and some commonly used ARM assembler respectively. Next we integrate the semantics for IMP with function call and the assembler and implement the semantics in $\mathbb{K}$ framework. We adopt it on running and verifying a sum program implemented in mixed language with the underlying interpretation and verification engine of $\mathbb{K}$. Furthermore, we conduct a case study on formally analyzing an industrial automobile operating system kernel to demonstrate the usefulness of our integrated semantics. To be more specific, our contribution can be summarized as:

- defining the semantics for IMP with function call and commonly used assembler, and integrating them together;
- implementing the integrated semantics in $\mathbb{K}$ framework and verifying a demo program implemented in mixed languages;
- formally analyzing the schedule mechanism of an industrial automobile operating system with our semantics.

The rest of this paper is organized as follows. Section

$$
\begin{array}{lrll}
word & \omega \in Int_{32} & & \\
Greg & r ::= & r_0 \mid r_1 \mid \ldots \mid r_{31} & \\
Sreg & sr ::= & CPSR \mid SPSR & \\
Addr & a ::= & \omega \mid [r] \mid [r_i] + [r_j] & \\
r_d & r_d ::= & Greg \mid Sreg & \\
r_s & r_s ::= & \omega \mid Greg \mid Sreg \mid a & \\
 & Instr ::= & STMDB\ sp,\ rs & \mid STMIB\ sp,\ rs \\
 & \mid & LDMIA\ sp,\ rs & \mid STR\ r_s,\ a \\
 & \mid & LDR\ r_d,\ a & \mid MOV\ r_d,\ r_s \\
 & \mid & ADD\ r_d,\ r_s,\ r_s & \mid SUB\ r_d,\ r_s,\ r_s \\
 & \mid & ORR\ r_d,\ r_s,\ r_s & \mid AND\ r_d,\ r_s,\ r_s \\
 & \mid & CMP\ r_s,\ r_s & \mid MRS\ r_d,\ sr \\
 & \mid & MSR\ sr,\ r_d & \mid B\ Id \\
 & \mid & BL\ Id & \mid BEQ\ Id
\end{array}
$$

Fig. 1. The syntax of the ARM assembly language

II defines the semantics for assembler and C respectively. Section III integrates the semantics of two languages. Section IV implements the semantics in $\mathbb{K}$ framework and applies the integrated semantics on verifying a program implemented in mixed language. Section V demonstrates how our approach can be adopted to verifying the interrupt mechanism of operating system kernel. Finally, we discuss some related work in Section VI and make a conclusion in Section VII.

II. Semantics for macro-ARM assembler and IMP with function call

This section defines semantics for the two languages respectively. Since our aim is to verify system kernels by defining semantics for its implementation language. In this paper we are caring a subset of ARM instructions which are frequently used in the operating system kernels. The communication between the two language programs are mainly by function calls. Thus we take IMP, and extend it with function calls, as the high level language in this paper.

A. Semantics for macro-ARM

The ARM instruction set provides programmers with a hardware-oriented assembly programming language. To formalize it, first we need to provide the abstract syntax of the given language. Then we define the configuration of the machine state. Finally, we give the operational semantics for the instructions.

Figure 1 shows the syntax of the ARM assembly language. Here we only give some typical instructions that are frequently used in OS implementations. $STMDB$ pushes the value of registers rs into the stack. The expression sp is the stack pointer, whose value is often stored in the register with index 13. rs is the register list. The instruction $STMIB$ is to increase the value of stack pointer and then store the values of registers rs into the stack. $LDMIA$ loads the value of the stack into the registers and then decreases the value of the stack pointer. The instructions STR and LDR are to load the value from memory with address a to a register r_s and store the value of register r_s to the memory address a. The instruction $MOV\ r_d,\ r_s$ is to move the value of register r_s to the destination register r_d. ADD (SUB) adds (subtracts) the value of two source operand and store the result to the destination operand. Similarly, ORR and AND are two opcodes which compute the logical or and logical and result of the two source registers and store the result to the destination register. Except for operating on general purpose register, there are some instructions that operate on status registers. For instance, MRS and MSR are two opcodes read value from status register to general purpose register and read value from general purpose register to the status register respectively. B and BL are to jump a specific address to execute the instructions. The difference is that the opcode BL should save its current program counter so that it could return after executing the called program.

We define the operational semantics for the instructions listed in Figure 1. The operational semantics specify how the effects of an instruction is produced. We firstly introduce the configuration that is used to describe the state of the assembly system, afterwards list the transition rule for all the instructions.

Configuration. The configuration of ARM assembler consists of ARM instruction set, a byte-addressable memory $Mem\ :\ N \mapsto B^8$ (where N is the natural number and $B \equiv \{0,\ 1\}$), two registers, i.e. general purpose register $GReg :: R \mapsto B^8$ and status register $SReg :: R \mapsto B^8$ (R is the register index and B^8 is the register value), and a data stack.

$$
\begin{aligned}
Con_{asm} = (&Pro_{asm} :: Instr,\\
&Mem :: N \mapsto B^8,\\
&GReg :: R \mapsto B^8,\\
&SReg :: R \mapsto B^8,\\
&Stack :: List)
\end{aligned}
$$

Semantics. For a better understanding of our semantics, we introduce some helper functions firstly. The function *value* is to read the value from a specific register or memory address.

The instruction LDR loads a value N to a register R_i. Executing the instruction $LDR\ R_i,\ N$ could change the state of general purpose register, i.e., replace the value of register R_i with N. After executing this instruction, the program component in the configuration becomes ε and waiting for the next instruction.

The instruction STR stores the value of register R_i to the memory address A_i. The execution of instruction STR changes the state of memory, i.e., replaces the data stored in the memory with address A_i with the value of register R_i. Similar with LDR, finishing executing this instructions means the program component becomes ε.

$STMDB$ reads values from general purpose registers and push the value into stack. In practice the stack pointer is the general purpose register with index of 13. For a clear understanding, our semantics regards the stack as an independent component. The execution of instruction $STMDB$ pushes the

TABLE I
SEMANTICS FOR ARM INSTRUCTIONS

Instruction	Semantics
LDR	$<(\textbf{LDR}\ R_i,\ N),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg[N/value(R_i)],\ SReg,\ Stack>$
STR	$<(\textbf{STR}\ R_i,\ A_i),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem[value(R_i)/value(A_i)],\ GReg,\ SReg,\ Stack>$
STMDB	$<(\textbf{STMDB}\ sp,\ N),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg,\ SReg,\ [N\cdot Stack/Stack]>$
STMIB	$<(\textbf{STMIB}\ R_i,\ \{R_j - R_n\}),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <STMIB\ [R_i]+4,\ \{R_{j+1}-R_n\}, Mem[value(R_j)/value(R_i)],\ GReg,\ SReg,\ Stack>$
LDMIA	$<(\textbf{LDMIA}\ sp,\ R_i),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg[N/value(R_i)],\ SReg,\ [N\cdot StackList/StackList]>$
ADD	$<(\textbf{ADD}\ R_i,\ R_{j1},\ R_{j2}),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg[(value(R_{j1})+value(R_{j2}))/value(R_i)],\ SReg,\ Stack>$
CMP	$<(\textbf{CMP}\ R_i,\ R_j,\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg,\ SReg[value(flag)/(value(R_i) == value(R_j))],\ stack>$
MRS	$<(\textbf{MRS}\ R_i,\ CPSR),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg[value(CPSR)/value(R_i)],\ SReg,\ Stack>$
MSR	$<(\textbf{MSR}\ CPSR,\ R_i),\ Mem,\ GReg,\ SReg,\ Stack> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg,\ SReg[value(R_i)/value(CPSR)],\ Stack>$
BL	$<(\textbf{BL}\ X,\ Mem,\ GReg,\ SReg,\ Stack,\ fun> \ \rightarrow\ <\pi_2.fun(X),\ Mem,\ GReg,\ SReg,\ Stack,\ fun>$
BEQ$_1$	$<(\textbf{BEQ}\ X,\ Mem,\ GReg,\ SReg,\ Stack,\ fun> \ \rightarrow\ <\pi_2.fun(X),\ Mem,\ GReg,\ SReg,\ Stack,\ fun>\ \ if\ value(flag) == 1$
BEQ$_2$	$<(\textbf{BEQ}\ X,\ Mem,\ GReg,\ SReg,\ Stack,\ fun> \ \rightarrow\ <\varepsilon,\ Mem,\ GReg,\ SReg,\ Stack,\ fun>\ \ if\ value(flag) == 0$

integer N into the stack.

$STMIB$ reads values from a list of registers and store them into the memory. The first memory address is the value of R_i. We regard its execution as non-atomoic. After one step of execution, this instruction is reduced to $STMIB\ [R_i] + 4,\ \{R_{j+1} - R_n\}$, where $[R_i] + 4$ means the start address has been changed. And since the value of R_j has been stored, the next value to be read is that of register R_{j+1}. Executing one step of this instruction replaces the value of address $value(R_i)$ in the memory with value of register R_i.

The instruction $LDMIA$ pops the value from the stack and moves the value to the specific register. The execution of this instruction triggers state transitions of two configuration components, i.e., taking out the first value N of the stack and replacing the value of register R_i with N.

ADD is a kind of arithmetic operator which adds the value of its two source operands and store the result in the destination register. As shown in the formula below, the execution of instruction ADD changes the state of the general purpose register, i.e., replace the value of R_i with the addition result of two source registers.

CMP is a logical operator which compares the values of its two source operands. If the their values are same, the flag bit of status register would be set with 1, otherwise 0 would be assigned to the flag bit of status register.

MRS is shorted for move to register from state register. It is used to operate on status registers. In another words, it replace the value of the destination register with that of CPSR, which stores the current program state.

MSR is shorted for move to state register from register. Same with MRS, it also operates on status registers by it replace the value of the CPSR with that of the source register.

The instruction BL jumps directly to a location indicated by the target label X if the target location is within the current procedure body. To integrate with C semantics, we get rid of the program counter in the ARM semantics. Instead, we regard a fragment separated by a label as a procedure, and jumping to a label is considered as a procedure call. Thus executing one step of this instruction is reduced to execute the function body with function name X. We omit the semantics for instruction B here because they are definitely same.

The instruction BEQ often follows some logical instructions, such as CMP. In case the flag bit of status instruction has been set with the value 1, the BEQ instruction has the same execution rule with BL, i.e., to call function X to execute. If the value flag bit is 0, the BEQ does nothing but just skip.

$stmt ::=\ Id = e \mid \texttt{if}\ e_1\ e_2\ \texttt{else}\ e_3 \mid \texttt{while}\ e_1\ e_2 \mid FunctionCall \mid \texttt{return}\ e$
$e ::= Int \mid Id \mid e_1 * e_2 \mid e_1 + e_2 \mid e_1 \leq e_2 \mid !e \mid e_1 \land e_2$
$FunctionCall ::=\ Id(e)$

Fig. 2. The syntax of IMP with function call

B. Semantics for IMP with function call

The syntax of C is presented in BNF form. e is defined as expressions, including arithmetic expression and boolean expression. It is clear that the symbols e_1 and e_2 are being used to stand for any arithmetic expression. In our presentation of syntax we use such metavariables to range over the syntactic sets-the metavariables e_1 and e_2 above are understood to range over the set of arithmetic expressions. The symbol "::=" should be read as "can be" and the symbol "|" as "or". Thus as arithmetic expression e can be an integer or an identifier, or (arithmetic or boolean) expression. As for the statements, we are concerning some basic statements, such as assignment statement, condition statement, loop statement and statements related to function call and function return.

With the syntax of IMP with function call, we define the transition rules for each statement to specify how the state changes caused by the execution of each statement. Similarly, the configuration should be defined firstly to describe the state of this language.

Configuration. An IMP with procedure call configuration is shown in the below, which consists of a global, byte-addressable memory $Mem :: N \mapsto B$, an environment $Env :: Id \mapsto N$, a function frame stores the defined functions and a function stack to store context when function call occurs.

$$Con_c = (Pro_c :: Stmts,\ Mem :: N \mapsto B,\ Env :: Id \mapsto N,\ Fun :: Id \mapsto (Paras,\ stmts),\ FStack :: List)$$

we present the situation of expression $e_1 \ * \ e_2$ waiting to be evaluated in the current state. To evaluate this expression, we should first get the values of e_1 and e_2 respectively. We assume both of those two expressions are identifier and the value can be presented as $Mem(Env(e_1))$.

To get the value of an identifier X, we should firstly look up the address of X by the function $Env(X)$, and then read the number store in address $Env(X)$ by the function $Mem(Env(X))$.

As an effect of such a statement execution the evaluated right hand expression of this statement is stored in the memory of the IMP with procedure call configuration at address that is the evaluated left hand side expression of this statement.

This statement performs a so-called conditional jump to the target expression. There are two cases for this expression: (i) if the evaluated logical test expression e_1 of the statement is equal zero, then it jumps e_2 to execute. (ii) in case the evaluated test expression e fails, i.e. it is not equal zero, then the execution of such a statement jumps to e_3 to execute.

The loop statement also could result in jumping to a target expression with conditions. If the boolean expression is evaluated as true, the loop expression jumps to e_2 to execute followed with the loop expression. In another case, if the boolean expression e_1 is evaluated as false, the loop expression does nothing.

A function call statement is treated here if it is not external and it either is a function or procedure call. The function to be called is e with parameters E. we should read the function body and its argument from the component fun to execute. As a result of the call statement execution a new stack frame is created and is put on the top of the stack. So that, the execution will resume at the next statement after the call statement whenever the called function(procedure) returns.

There are two return statements: (i) return from a function call, i.e. return with result, and (ii) return from a procedure call, i.e. return without result. In the first case the evaluated result expression of the return statement is stored in the memory at the address identified by the return destination component. For both statements the top stack frame in the configuration is removed.

III. Integrate IMP-FC and assembler Semantics

We extend both the IMP with function calls and assembler by allowing the IMP-FC programs call assembler functions and assembler programs call IMP-FC functions. Besides, the operand of assembler instructions could be a C variable. Based on the extension of the two languages, we integrate the semantics for them in this section.

A. Semantics of the Integrated language

In order to obtain an integrated model of IMP with procedure call and assembly language, there are two things left to do: define how we model the state of the combined semantics and define transitions. **Configuration**

C=$(Pro_{asm} \cup Pro_c,\ Mem,\ Con_{asm},\ Con_c)$

$Con_{asm} = (Greg,\ Sreg,\ stack)$

$Con_c = (env,\ Fstack,\ Fun)$

$\pi_1.con_{asm} =_{df} Greg \qquad \pi_1.con_c =_{df} env$

$\pi_2.con_{asm} =_{df} Sreg \qquad \pi_2.con_c =_{df} fstack$

$\pi_3.con_{asm} =_{df} stack \qquad \pi_3.con_c =_{df} fun$

The program component of integrated language is simply a record consisting of an IMP with procedure call program and assembly program. Another observation that can be made is that in both semantics we use the same byte-addressable memory, which can be shared. In order to eliminate redundancy, we introduce the notion of execution context for IMP and assembly in the integrated semantics. A context is a configuration of the corresponding language where the program and memory component (Pro and Mem) are removed. Each context is a snapshot and can be expressed by a triple. We select the components of a snapshot using the projections, where π_n is to get the n^{th} element of the snapshot.

For an external call from IMP to assembly, the context of assembly should be updated according to the calling convention, i.e., passing the parameter values to the first four registers in case the amount of parameters is not greater than four. The currently active IMP context is retired to the list of inactive contexts. And the body of function e is taken out to exexute.

For a call from assembly to IMP programs, we should assign values to the parameters of IMP functions firstly. The first four parameters are taken from registers, we convert their values to C-IL-values of the type expected by the function. The remaining parameters are passed on the stack (lifo) in right-to-left order. Then we take out the function body of function e from the fun component, which is the third element of IMP context, to execute.

TABLE II
SEMANTICS FOR IMP WITH FUNCTION CALL

statements	semantics
arithmetic	$< e_1 \ * \ e_2, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < Mem(Env(e_1)) * Mem(Env(e_2)), \ Mem, \ Env, \ Fun, \ Fstack >$
look up	$< X : Id, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < Mem(Env(X)), \ Mem, \ Env, \ Fun, \ Fstack >$
assignment	$< X = n, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < \varepsilon, \ Mem[n/Mem(Env(X))], \ Env, \ Fun, \ Fstack >$
condition$_1$	$< if \ e_1 \ then \ e_2 \ else \ e_3, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < e_2, \ Mem, \ Env, \ Fun, \ Fstack > \quad if \ Mem(Env(e_1)) == true$
condition$_2$	$< if \ e_1 \ then \ e_2 \ else \ e_3, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < e_3, \ Mem, \ Env, \ Fun, \ Fstack > \quad if \ Mem(Env(e_1)) == false$
loop$_1$	$< while \ e_1 \ e_2, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < e_2; \ while \ e_1 \ e_2, \ Mem, \ Env, \ Fun, \ Fstack > \quad if \ Mem(Env(e_1)) == true$
loop$_2$	$< while \ e_1 \ e_2, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < \varepsilon, \ Mem, \ Env, \ Fun, \ Fstack > \quad if \ Mem(Env(e_1)) == false$
function call	$< e(E), \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < \pi_2.fun(X)[E/\pi_1.Fun(e)], \ Mem, \ \varepsilon, \ Fun, \ [(C.Proc_c, \ Env) \cdot Fstack)/Fstack] >$
function return	$< return \ e, \ Mem, \ Env, \ Fun, \ Fstack > \ \to \ < Mem(Env(e)) \circ s_1, \ Mem, \ Env_1, \ Fun, \ [stackList/(S_1, \ Env_1) \cdot stackList] >$

BL is another instruction that can call IMP functions from assembler. The difference with instruction *B* is *BL* is to call functions with return. That is to say before calling functions, the execution context of assembly program should be stored and it will be restored after executing function *e*. The state transition is exactly same with that of instruction *B*, we omit its explanation here.

The return statement performs an interlanguage return such that the IMP execution context that has created the current active assembly context becomes active again. This is done by substituting the current active assembly context with the IMP execution context found on the top of inactive execution contexts.

This case is complementary to the previous one. If an assembly function returns to IMP, the execution context of IMP stored in the function stack should be taken out. That is to say, to restore the IMP statements *stmt* to the program component and the *env* to the environment component.

Except for function calls, the assembler programs and IMP programs can interactive by accessing C variables. There are two cases where assembler programs accesses IMP variables: (i) read the address or the value of IMP variables; (ii) take the IMP variables as an operand of assembly instructions. In the second case, we obtain the value of the variable firstly and take the integer as one of the operands to execute the instruction.

IV. SIMULATION AND APPLICATION OF THE INTEGRATED SEMANTICS

We implement our semantics in 𝕂-a rewrite-based semantic framework in this section. The implementation mainly consists of the syntax of the integrated language, the configuration and the semantics for each statement and instructions. Besides, to demonstrate the effectiveness of the semantics, we applied it on verifying a demo mixed language program in this section.

A. 𝕂 Framework

𝕂 is a semantic definitional framework based on rewriting logic. Rewriting logic [11] is a computational model for concurrency, distributed algorithms, programming languages, software and hardware systems. It can also be regarded as a logic for executable specification and analysis. Many languages and systems based on rewriting logic are designed and implemented, such as CafeOBJ [12], ELAN [13], Maude [14] and 𝕂, among which, 𝕂 is one of the state-of-the-art frameworks with features of simplicity, executability, analysability, and scalability [15].

𝕂 framework is used to analyse the program languages based on their formal semantics. The syntax of language is defined using BNF (Backus-Naur Form) in 𝕂. And the semantic definitions are like operational semantics, which denote the transition rules of the system. 𝕂 framework integrates many capabilities of program analysis. Given a syntax and a semantics of a language, 𝕂 generates a parser, an interpreter, a model checker through its Maude backend and a deductive theorem prover by translating its semantics into Coq definitions. The parser and interpreter allow a program to execute based on its semantics. Furthermore, the 𝕂 can analyse the program more comprehensively than compilers. For example, the model checking capability helps the language designer to cover all the non-deterministic behaviors of certain language through the state-space exploration. The main advantages of 𝕂 are:

- 𝕂 supports modular definition of formal semantics and user-defined data types.
- The formal semantics defined in 𝕂 is executable, which provides not only a way to formally analyse programs but a way to test the correctness of defined semantics.
- 𝕂 framework has many backend formal tools for interpreting programs, model checking and theorem proving.

𝕂 is built upon three main components, which are configuration, computational rules and structural rules. The configuration is a nested cell to present the static state of a program being executed. The basic ingredient of a 𝕂 definition is cell, which is labeled and stands for a piece of information of a state. The 𝕂 rules are executed by changing the information of states stored in cells. Both computational rules and structural rules are 𝕂 rules. The difference is that computational rules are counted as computational steps while structural rules are to rearrange the structure of configuration.For better understanding of rewrite rules, consider the following example:

$$\langle\langle \$PGM : pgm\rangle_k \ \langle .Map\rangle_{env}\rangle_T \qquad [\text{configuration}]$$

$$\langle \frac{X = I;}{.} \ldots\rangle_k \ \langle X \mapsto \frac{-}{I} \ldots\rangle_{env} \qquad [\mathbb{K}\ \text{rule}]$$

The illustrated example shows the 𝕂 definition of a configuration and a 𝕂 rule. The configuration only contains two cells,

TABLE III
Semantics for Integrated Language

statements	semantics
function call (C)	$< Pro_c = e(E),\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow$
	$< \pi_2.\pi_3.conc_c(e)[E/value\{R_0 - R_3\}],\ Mem,\ Con_{asm},\ \ Conc_c[(next.Pro_c,\ \pi_1.Con_c)\cdot\pi_2.Con_c/\pi_2.Con_c,\ \varepsilon/\pi_1.Con_c] > \quad if\ \lvert E\rvert \le 3$
function call (asm1)	$< Pro_{asm} = B\ e,\ Mem,\ Con_{asm},\ Con_c > \rightarrow\ < \pi_2.\pi_3.Conc_c(e)[valueR_0 - R_3/\pi_1.\pi_3.Conc_c(e)],\ \ Mem,\ Con_{asm},\ Con_c > \ \ if\ \lvert\pi_1.\pi_3.Conc_c(e)\rvert \le 3$
function call (asm2)	$< BL\ e,\ Mem,\ Con_{asm},\ Con_c > \quad \rightarrow\ < \pi_2.\pi_3.Conc_c(e)[valueR_0 - R_3/\pi_1.\pi_3.Conc_c(e)],\ \ Mem,\ Con_{asm},\ Con_c > \quad if\ \lvert\pi_1.\pi_3.Conc_c(e)\rvert \le 3$
return (C)	$< Pro_c = return\ e,\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow\ < \varepsilon,\ Mem,\ Con_{asm}[\pi_1.Con_{asm}[e/R_0]/\pi_1.Con_{asm}],\ Con_c >$
return (asm)	$< Pro_{asm} = END,\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow\ < stmt_1,\ Mem,\ Con_{asm},\ env_1/\pi_1.Con_c],\ \ Conc_c[\pi_2.Con_c'/((stmt_1,\ env_1)\cdot\pi_2.Con_c') >$
global variable	$< Pro_{asm} = (OP\ R_i,\ R_j,\ A_i),\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow\ < (OP\ R_i,\ R_j,\ Mem(A_i)),\ Mem,\ Con_{asm},\ Con_c >$

one for computation and another for storing the data states of variables. The *k* cell represents a list (or stack) of computations waiting to be performed. The left-most (i.e., top) element of the stack is the next item to be computed. The *env* cell is a map of variables to their locations.

The second line presents an example of 𝕂 rule, which defines the semantics of assignment operation. The fraction-style term in cells indicates the transition from the value over the bar to the one below it. The cells without horizontal lines are only read but not changed by the rule. When the next statement to be executed is to assign the variable *X* with value *I*, the map value of variable *X* should be updated as *I*. In *env*, the "-" is an anonymous variable, which can match the original value in the corresponding location, and the horizontal line indicates a reduction. The statement $X = I$; is consumed, as represented by "." which stands for the unit computation. The "..." are structural frames to match the irrelevant portions in this cell.

B. Implement the Semantics in 𝕂 framework

We give practical insights into the 𝕂 framework by defining the integer subset of the ARM assembly language. Whereas we defined the entire subset of integer-based instructions, IMP with function calls and the integrated semantics, for brevity we only describe a representative snippet of it. The general methodology for language definitions in 𝕂 begins with the (abstract) syntax, determines the configuration, and then gives the semantic rules. The program configuration is wrapped multiset of cells, written as $< cont >_{lbl}$, , where cont is the cell contents (possibly itself a multiset of cells) and lbl is the cell label. The 𝕂 cells hold the necessary semantic infrastructure (registers, instruction cache, memory, etc.). Two cells appear in most 𝕂 definitions: a cell whose label is ⊤ that encloses all the other cells, and a cell labeled k that holds the computation. In this section, we present how the semantics of assembly and integrated language can be implemented in 𝕂. The 𝕂 semantics of IMP with function call can be defined in the similar way and we omit the details here.

1) 𝕂 Semantics of Assembly: We have implemented the semantics for all the ARM instructions listed in TABLE I in 𝕂, Figure 3 lists some of them. We take four semantic rules as example to illustrate how we define the execution rules for the memory operation, arithmetic expression, logical expression and jump instruction respectively.

rule STORE

$$\langle \frac{STR\ \ r\ \ I,\ X}{.K} ... \rangle_k \quad \langle ...I|- > I1... \rangle_{reg}$$
$$\langle ...X|- > L... \rangle_{env} \quad \langle ...L|- > \frac{=}{I1} ... \rangle_{store}$$

rule ADD

$$\langle \frac{ADD\ \ r\ \ I,\ \ I2,\ \ I3}{.} ... \rangle_k \quad \langle \frac{R}{R[I{<}{-}(I2\ \ {+}Int\ \ I3)]} \rangle_{reg}$$

rule CMP

$$\langle \frac{CMP\ \ r\ \ I,\ \ I2,\ \ I3}{.} ... \rangle_k \quad \langle ...I|- > I2... \rangle_{reg}$$
$$\langle \frac{R}{R[0{<}{-}(I2\ \ {==}Int\ \ I1)]} \rangle_{sreg}$$

rule B

$$\langle \frac{B\ \ X{\sim}{>}K}{Is} \rangle_k \quad \langle ...X|- > functionBody(Is)... \rangle_{fun}$$

Fig. 3. Semantics for Assembly

The program to be executed should be stored in the *k* cell. In case the next instruction to be executed is $STR\ rI,\ X$, it should be the top of the computation stack. After execution, this instruction has been consumed and reduced to empty, which is represented as $.K$ in the 𝕂 semantics. The ... in *k* cell means there may be some instructions left to be executed but we do not care about them right now. STR is to assign the value stored in the register with index *I* to the variable *X*. Thus we have to located the address of variable *X* from the cell *env*, and then store the value of register *I* to the corresponding address.

The execution rule of instruction *ADD* and *CMP* the quite similar. Before execution the instruction, we have read the values of the source operands and replaced the register indexes with their values. Then the execution of *ADD* (or other arithmetic expressions) is to store the addition (or the corresponding calculation) of the two operands to the destination register. Similarly, the execution of CMP (or other logic expression) is to store the comparison (or the corresponding logic operation) result to the flag bit of status register.

rule B

$\langle \frac{B\ \ X{:}Id \curvearrowright \_}{AssPara(Ps,\ 0) \curvearrowright Ss} \rangle_k \quad \langle \frac{.List}{ListItem((.K))} \cdots \rangle_{fstack}$

$\langle ...\ X\ |-> functionBody(Ps : Parems,\ \_) : Type,\ Ss : Stmts) ... \rangle_{fun}$

rule BL

$\langle \frac{B\ \ X{:}Id \curvearrowright \_}{AssPara(Ps,\ 0) \curvearrowright Ss} \rangle_k \quad \langle \frac{.List}{ListItem(K)} \cdots \rangle_{fstack}$

$\langle ...\ X\ |-> functionBody(Ps : Parems,\ \_) : Type,\ Ss : Stmts) ... \rangle_{fun}$

rule AssCall

$\langle \frac{F1{:}Id(As{:}AExps) \curvearrowright K}{AssReg(As,\ 0) \curvearrowright Is \curvearrowright END} \rangle_k \quad \langle \frac{.List}{ListItem((Env,\ K))} \cdots \rangle_{fstack}$

$\langle ...\ F1\ |-> functionBody(Is) ... \rangle_{fun} \quad \langle \frac{Env{:}Map}{.Map} \rangle_{env}$

rule IMPReturn

$\langle \frac{return\ \ I{:}Int; \curvearrowright \_}{ldr\ \ r0,\ I \curvearrowright K} \rangle_k \quad \langle \frac{ListItem(K)\ \ L}{L} \rangle_{fstack}$

rule AssReturn

$\langle \frac{END \curvearrowright}{I \curvearrowright K} \_ \ {}_k \quad \langle \frac{\_}{Env} \rangle_{env}$

$\langle ...\ 0\ |->\ I\ ... \rangle_{reg} \quad \langle \frac{ListItem((Env,\ K))}{.List} \cdots \rangle_{fstack}$

Fig. 4. Semantic rules for integrated language

Some instruction fragment are labeled by an identifier (represented as I). We regard such an instruction fragment as a function with name I and put it into the function stack. As for the jump instruction $B\ X$, it can be considered as a function call instruction which is calling the function with name X. The execution of $B\ X$ is to take out the function body IS of X from the function stack, as shown in the last rule of Figure 3.

2) $\mathbb{K}$ Semantics of the Integrated language: We implement the semantics of integrated language in $\mathbb{K}$ as shown in Figure 4, including the transition rules for function calls and function returns. Basically, the execution rules for instructions B and BL are same. Both of them are reduced to a helper function $AssPara(Ps,\ 0)$, which is to pass the value of registers to the parameters of the function. Besides, the function body of function X is added to the computation cell to execute after passing the parameters.

In case of calling assembly functions by IMP programs, the function call statement $F1 : Id(As : AExps)$ with its execution context are reduced to a helper function $AssReg(As,\ 0)$, which is to pass the parameters to the first four registers. The instructions labelled by F1 are to be executed after passing parameters. Furthermore, the execution and environment context should be added into the cell of $fstack$.

The last two semantic rules are about function return, i.e., return from IMP function to the assembly program and return from the assembler function to IMP programs. In case of returning to assembly program from IMP, the return value should be stored to the first register (shown as $ldr\ r0,\ I$) and the execution context K of the calling function should be restored. Similarly, the case of returning from assembler function to IMP is also asked for restore the execution context and get out the value of the first register.

C. Application of the formal semantics

One particularly useful formal analysis tool developed for $\mathbb{K}$ is the Reachability Logic prover. This prover accepts as input a $\mathbb{K}$ definition and a set of logical reachability claims to prove. The prover then attempts to automatically prove the reachability theorems over the language's execution space, assuming

The $\mathbb{K}$ prover accepts reachability claims in the same format as semantic rules. Instead of interpreting the supplied module as axioms (like the modules in the semantics itself), the module is interpreted as a set of reachability claims.

We adopt the verifier of $\mathbb{K}$ in verifying programs implemented in integrated language. Suppose we want to sum the numbers from 1 to s by the function $loop(s)$ and then return the result to the IMP program. The main function is implemented in IMP while the $loop$ function is implemented in assembler.

```
int main()
{
int s;
int result;
s=10;
result=loop(s);
return result;}
_asm_{
loop
ADD r 2, r 2, #1
ADD r 1, r 1, r 2
CMP r 0, r 2
BEQ END
B loop
END}
```

The property we verify is that the sum from 1 to S is calculated correctly by executing the program. This is expressed with the following reachability claim:

```
rule
<k>
loop(S:Int)=>S...</k>
<env>...ENV:Map...</env>
<store>...STORE:Map...</store>
<fun>...FUN:Map...</fun>
<reg>.Map=>Reg:Map</reg>
<sreg>.Map=>Sreg:Map</sreg>
requires S>=Int 0
```

This states that the statement for calling function *loop* with parameter S is to be executed. The right hand side of the *k* cell states that the return value of function should be S. The elements in cells $< env >$, $< store >$ and $< fun >$ are to be read but will not be modified in the verification. Since the function *loop* is implemented by assembly code, the cells $< reg >$ and $< sreg >$ would be modified while we do not care about the exact information of them.

As the program has a loop, we need to supply a circularity which helps reason about how the remainder of the program behaves after any iteration of the loop. Note this is not quite the same as an invariant stating that each iteration of the loop maintains the partial sum. Reachability Logic allows specifying the traditional Hoare Logic rule as a loop invariant and performing proofs that way, but specifying circularities directly instead can be more intuitive and robust to changes in the program.

```
rule
<k>
add r 2, r 2, #1
add r 1, r 1, r 2
cmp r 0, r 2
BEQ END
B loop
END
=>
END
</k>
<fun>...FUN:Map...</fun>
<reg>
...
0 |-> S:Int
1 |-> (S1:Int=>S1 +Int I *Int (I +Int 1)/Int 2)
2 |-> (I:Int=>S)
...
</reg>
<sreg>...0 |-> (_=>true)...</sreg>
requires S>=Int 0
```

The cell $< k >$ stores instructions of the loop and *END* means the execution gets out of the loop. S is the parameter passed from IMP program, which is stored in the first register indexed by 0. S1 is the partial sum so far and I is the next number to be added. The result is stored in register $r1$, whose final value is $S1 + IntI * Int(I + Int1)/Int2$.

Given these two rechability claims, the $\mathbb{K}$ verifier verifies that this integrated-language executes correctly. The verification result also demonstrates the soundness of our semantics to some extend.

V. Automatic Verification of Multi-language Program

Our semantics has some industrial value in verifying some embedded systems. In processors of ARM7 architecture, the interruption mechanism is implemented in mixed language, i.e., C and assembly language.

A. The interruption mechanism of ARM ISA

The implementation of an OS kernel asks to disable and enable interrupts in some special cases. For instance, when scheduling happens disabling interrupt is needed in the process of selecting the next running task (or process). The interrupt control bit is stored in the register CPSR. The register can only be accessed by assembly instructions and the scheduling mechanism is often implemented in C to avoid redundancy.

The function of disabling interrupt is to store the value of register CPSR in the stack. And then set the interrupt control bit with 1, which means masking the interrupt. In another case, enabling interrupt is to clear the interrupt control bit. The implementation of disabling and enabling interrupt is listed in the following.

```
DisableAllInterrupts
      mrs r7, CPSR
      orr r7, r7, #0xC0
      msr CPSR, r7

END
```

```
EnableAllInterrupts
      mrs r7, CPSR
      and r7, r7, #0xFFFFFF3F
      msr CPSR, r7
END
```

The above two interrupt functions are often called in pairs when atomic operation is needed. For instance, an atomic operation context switch should take place when the system call schedule is called, which means the functions *DisableAllInterrupts* and *EnableAllInterrupts* are being called in case of performing scheduling. As we know, the *schedule* system call is often implemented in high level language (i.e., C language) except the operations which need to access registers. Thus the implementation of *schedule* is in mixed language. We take this system call as an example to demonstrate how our semantics can be applied to verify operating system kernels.

B. Verifying the interruption mechanism

```
schedule()
{
DisableAllInterrupts();
if (OSReadyQueue.First.Prio>OSRunningTcb.
    Prio)
{
nextTask.Tcb=OSReadyQueue(First.Tcb);
OSReadyQueueRemove(First.Id, First.Prio);
OSReadyQueueAppend(osRunningTask,
    osRunningTcb.Prio);
OSRunningTcb=nextTask.Tcb;
}
else{}
EnableAllInterrupts();
}
```

We looked into the implementation of an industrial OSEK-based operating system (OSEK OS for abbreviation) [16] kernel, whose formal semantics of system calls have been defined in [17]. The schedule mechanism of OSEK OS is based on priority scheduling algorithm, which means the scheduler always allocate the processor to the task with the highest priority. To remain the tasks in ready queue unchanged, the system call *schedule*() should disable interrupts before choosing the next allocated task and enable the interrupts after assigning the processor to the task.

We integrate our semantics with the semantics defined in [17] to apply our semantics on verifying operating systems. since we have only defined semantics for some basic C statements exclude pointers, arrays and some other complex data structures at present, we regard the statement with such data structures as a special statement and define its execution rule in our semantics. The implementation of the OSEK-based system call *schedule*() is shown below.

```
rule
<k>
if (OSReadyQueue.First.Prio>OSRunningTcb.Prio)
{
nextTask.Tcb=OSReadyQueue(First.Tcb);
OSReadyQueueRemove(First.Id, First.Prio);
OSReadyQueueAppend(osRunningTask,
                   osRunningTcb.Prio);
OSRunningTcb=nextTask.Tcb;
}
else{}
=>.K
...</k>
<store>...ENV:Map...</store>
<sreg>SREG:Map</sreg>
<readyTasks>
(T, N) L => add(T1, N1, L)
</readyTasks>
<runningTask> T1:Id => T </runningTask>
<store>...STORE:Map...</store>
requires N >Int N1
```

As for the system service *schedule*(), we should guarantee that there is no interruption in the process of context switch so that the states of tasks could not be changed. The property of *schedule*() is specified is specified in the below. First of all, we should make sure that the state of status register can not be changed (as shown in the cell $< sreg >$), which means interruption can not happen. Besides, in case there is a task T with a higher priority in the ready task list, the processor will be allocated to T and put the previous task to the ready list (as shown in the cell $< readyTasks >$ and $< runningTask >$). We only list the specification for the case when a task is allocated with a processor here, while we have also specified another case where the running task has the highest priority and no state transition occurs in our project.

VI. Related work

In this sections, we discuss some related works in three aspects, i.e. the integrated semantics for mixed language, semantics defined in $\mathbb{K}$ framework and the research on verifying OS kernels.

A. *integrated semantics*

Several projects have developed integrated semantics for mixed language programs. Schmaltz [1] et.al integrate the semantics macro-assembler and C immediate language. Their work mainly focuses on the function call and function return between two languages but ignores the case when assembler instructions accesses C variables. In the dissertation [18] Shadrin also presenting a method to integrate C and assembler together and complete paper-and-pencil formal proof of a small hypervisor. Following the work [18], Pentchev [2] adds the semantics for interrupt of operating system kernel to the integrated semantics defined in [18]. Moreover, the author verifies the interrupt mechanism with a C verifier (i.e., VCC) [19] by abstracting the hardware implementation to C level.

Compared to the above works, the semantics in this paper not only define semantics for function calls and function returns between the two languages but also take global variables, which could be accessed by both assembler and high level language, into account. Furthermore, we implement our semantics in a semantic framework $\mathbb{K}$, which helps us verify programs automatically.

B. $\mathbb{K}$ *semantics*

$\mathbb{K}$ has been deployed to defined semantics for a certain number of languages. C semantics in $\mathbb{K}$ (KCC) is the first complete language semantics defined in $\mathbb{K}$ [7]. KCC passed 99.2% of GCC torture test suit while GCC could only pass 99% of it. In the process of defining C semantics, some ambiguous and undefined interpretations in C standard were detected, which have been defined in the later work [20]. Thanks to the analyzability of $\mathbb{K}$, KCC has been commercialized to verify the correctness of C implementations.

$\mathbb{K}$ also provides executable semantics for java [8], javascript [9], php [21], rust [22], etc. Besides programming languages, the $\mathbb{K}$ framework defines formal virtual machines for various software systems, such as KEVM [23] for Ethereum Virtual Machine [24] and IELE [25], [26] for blockchain system. Except for program languages, $\mathbb{K}$ has proven its scalability and effectiveness by defining semantics for software systems. For instance, [17] defined semantics for system services of operating system kernel and verifying operating system applications. [27] deployed $\mathbb{K}$ framework to formally verify the Orc orchestration, which is a concurrent computation model describing the composition and management of web services.

To our best knowledge, the current $\mathbb{K}$ semantics are about single language, while we use $\mathbb{K}$ to bridge the gap between two languages. Furthermore, we demonstrate that $\mathbb{K}$ has the capability of verifying a mixed language program.

C. OS kernel verification

Operating system kernel verification has attracted considerable attention for a long time [28]. SeL4 [29] is considered as the first fully verified OS kernel. It is verified by the interactive theorem prover HOL/Isabelle [30] and the project takes more than 20 person years. Besides, Gu. et al [31] use interactive theorem prover Coq [32] to verify the correctness of a concurrent kernel in multi-core at assembly level. [33] combines relyguarantee-based simulation with the program logic for reasoning about interrupts to verify key modules in the C/OS-II kernel in Coq. All these impressive achievements need construct a machine-checkable proof and come with a non-trivial cost. In our work, the mixed language programs could be verified automatically thanks to the Z3-based [34] verifying engine provided by $\mathbb{K}$.

To reduce the verification efforts, automatic OS kernel verification has long been a research objective. [35] translates both C and assembler implementations to LLVM [36] and adopts the LLVM verifier to verify the correctness of the OS kernel. [37] abstracts the assembler programs to C level and verify the OS kernel with the C verifier. [38] compiles C to assembler and conducts all verification at assembly level with Dafny [39]. The verifier deployed by the above works are all based Z3, which is also used by $\mathbb{K}$. Compared with these works, our approach needs no translation of the program. In this case, developers do not need to learn a new language to verify the OS kernel. On the other side, keeping the programs untouched would not affect the readability of the system implementation, which is easier to specify properties for the system.

Interrupt mechanism of OS kernel is an interesting topic that attracts many researchers. [40] proposes an interrupting modeling language, and defines the operational semantics and denotational semantics for it to analyze systems with interruption. Compared with this work, our approach does not need to define denotational semantics and can verify the interrupt mechanism automatically. [41] verifies the properties of interrupt mechanism from binary code level by applying PRISM. The basic idea of this work is to abstract an particular interrupt mechanism as a discrete-Time markov chain, which is not as general as the approach proposed in this paper.

VII. Conclusion and Future Work

This paper presented an integrated semantics for assembler and a high level language. The integrated semantics not only take function calls and function returns between the two languages into account, but also specified the transaction rules for the cases when a global variable could be accessed by different languages. We adopted an executable semantic framework $\mathbb{K}$ to implement our semantics and verified the correctness of a mixed language program. To demonstrate the effectiveness of our semantics, we applied it to verifying the schedule mechanism of an industrial operating system kernel.

As future work, we are considering to apply our semantics in verifying some critical parts of operating system kernels, such as memory allocation and context switch, which are implemented in mixed language. To achieve this goal, we have to complete our semantics for high level language since the implementation of operating system kernel involves many complex data structures, such as pointers and lists. Thanks to the $\mathbb{K}$ semantics of C, it can be attained by integrating $\mathbb{K}$ semantics for C and our semantics.

Another promising direction we intend to pursue is to verify operating system applications. The basic idea is to specify a rather precise formal semantics for system calls defined in the operating system kernel. [17] has defined semantics for system calls of OSEK/VDX operating system kernel. However, they regard s each system call as an atomic statement while in practice the execution of a system call be interrupted by the environment or alarms. We are planing to refine their semantics by considering interruptions (i.e., splitting the semantics for system calls at the points where disabling and enabling interrupts are called). In this case, the verification for operating system applications could be more circumspect.

References

[1] S. Schmaltz and A. Shadrin, "Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from verisoftxt," in *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, 2012, pp. 18–33. [Online]. Available: https://doi.org/10.1007/978-3-642-27705-4_3

[2] H. Pentchev, "Sound semantics of a high-level language with interprocessor interrupts," Ph.D. dissertation, Saarland University, Saarbrücken, Germany, 2016. [Online]. Available: http://scidok.sulb.uni-saarland.de/volltexte/2016/6556/

[3] G. Rosu, "K - a semantic framework for programming languages and formal analysis tools," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security, D. Peled and A. Pretschner, Eds. IOS Press, 2017.

[4] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[5] A. J. Summers and P. Müller, "Automating deductive verification for weak-memory programs," *CoRR*, vol. abs/1703.06368, 2017. [Online]. Available: http://arxiv.org/abs/1703.06368

[6] S. K. Lahiri, A. S. Murawski, O. Strichman, and M. Ulbrich, "Program equivalence (dagstuhl seminar 18151)," *Dagstuhl Reports*, vol. 8, no. 4, pp. 1–19, 2018. [Online]. Available: https://doi.org/10.4230/DagRep.8.4.1

[7] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, 2012, pp. 533–544. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103719

[8] D. Bogdanas and G. Rosu, "K-java: A complete semantics of java," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, 2015, pp. 445–456. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676982

[9] D. Park, A. Stefanescu, and G. Rosu, "KJS: a complete formal semantics of javascript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 346–356. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737991

[10] C. Ellison, "A formal semantics of C with applications," Ph.D. dissertation, University of Illinois, July 2012.

[11] S. Escobar, "Rewriting logic and its applications (extended selected papers from WRLA 2014)," *J. Log. Algebr. Meth. Program.*, vol. 86, no. 1, pp. 157–158, 2017. [Online]. Available: https://doi.org/10.1016/j.jlamp.2016.10.002

[12] R. Diaconescu and K. Futatsugi, "An overview of cafeobj," *Electr. Notes Theor. Comput. Sci.*, vol. 15, pp. 285–298, 1998. [Online]. Available: https://doi.org/10.1016/S1571-0661(05)80017-9

[13] P. Borovanský, C. Kirchner, H. Kirchner, and P. Moreau, "ELAN from a rewriting logic point of view," *Theor. Comput. Sci.*, vol. 285, no. 2, pp. 155–185, 2002. [Online]. Available: https://doi.org/10.1016/S0304-3975(01)00358-9

[14] J. Meseguer, "Maude," in *Encyclopedia of Parallel Computing*, 2011, pp. 1095–1102. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_248

[15] T. F. Şerbănuţa, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu, "The K primer (version 3.2)," Tech. Rep.

[16] OSEK Group *et al.*, "OSEK/VDX operating system specification," 2009.

[17] 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.

[18] A. Shadrin, "Mixed low- and high level programming language semantics and automated verification of a small hypervisor," Ph.D. dissertation, Saarland University, 2012. [Online]. Available: http://scidok.sulb.uni-saarland.de/volltexte/2012/4964/

[19] 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, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, 2009, pp. 23–42. [Online]. Available: https://doi.org/10.1007/978-3-642-03359-9_2

[20] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of C," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 336–345. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737979

[21] D. Filaretti and S. Maffeis, "An executable formal semantics of PHP," in *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, 2014, pp. 567–592. [Online]. Available: https://doi.org/10.1007/978-3-662-44202-9_23

[22] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang, "Krust: A formal executable semantics of rust," *CoRR*, vol. abs/1804.10806, 2018. [Online]. Available: http://arxiv.org/abs/1804.10806

[23] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the ethereum virtual machine," in *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, 2018, pp. 204–217. [Online]. Available: https://doi.org/10.1109/CSF.2018.00022

[24] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014.

[25] T. Kasampalis, D. Guth, B. Moore, T. Serbanuta, V. Serbanuta, D. Filaretti, G. Rosu, and R. Johnson, "Iele: An intermediate-level blockchain language designed and implemented using formal semantics," University of Illinois, Tech. Rep. http://hdl.handle.net/2142/100320, July 2018.

[26] Runtime Verification Inc., "IELE: A new virtual machine for the blockchain," https://runtimeverification.com/blog/iele-a-new-virtual-machine-for-the-blockchain, accessed May 18, 2018.

[27] M. A. AlTurki and O. Alzuhaibi, "Towards formal verification of orchestration computations using the K framework," in *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, 2015, pp. 40–56. [Online]. Available: https://doi.org/10.1007/978-3-319-19249-9_4

[28] Y. Zhao, Z. Yang, and D. Ma, "A survey on formal specification and verification of separation kernels," *Frontiers Comput. Sci.*, vol. 11, no. 4, pp. 585–607, 2017. [Online]. Available: https://doi.org/10.1007/s11704-016-4226-2

[29] 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. [Online]. Available: http://doi.acm.org/10.1145/1743546.1743574

[30] T. Nipkow and G. Klein, *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. [Online]. Available: https://doi.org/10.1007/978-3-319-10542-0

[31] 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, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 653–669. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[32] J. Cao, M. Fu, and X. Feng, "Practical tactics for verifying C programs in coq," in *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, 2015, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/2676724.2693162

[33] 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, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 59–79. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_4

[34] N. Bjørner, "Z3 and SMT in industrial r&d," in *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, 2018, pp. 675–678. [Online]. Available: https://doi.org/10.1007/978-3-319-95582-7_44

[35] 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, Shanghai, China, October 28-31, 2017*, 2017, pp. 252–269. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132748

[36] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[37] J. Ding, X. Zhu, and J. Guo, "End-to-end automated verification for os kernels," in *25th Asia-Pacific Software Engineering Conference, APSEC, 4-7 December 2018, Nara, Japan, 2018*, 2018.

[38] 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, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 165–181. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[39] P. Lucio, "A tutorial on using dafny to construct verified software," in *Proceedings XVI Jornadas sobre Programación y Lenguajes, PROLE 2016, Salamanca, Spain, 14-16th September 2016.*, 2016, pp. 1–19. [Online]. Available: https://doi.org/10.4204/EPTCS.237.1

[40] Y. Huang, J. He, H. Zhu, Y. Zhao, J. Shi, and S. Qin, "Semantic theories of programs with nested interrupts," *Frontiers Comput. Sci.*, vol. 9, no. 3, pp. 331–345, 2015. [Online]. Available: https://doi.org/10.1007/s11704-015-3251-x

[41] J. Shi, L. Zhu, Y. Huang, J. Guo, H. Zhu, H. Fang, and X. Ye, "Binary code level verification for interrupt safety properties of real-time operating system," in *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, 2012, pp. 223–226. [Online]. Available: https://doi.org/10.1109/TASE.2012.46