Formal semantics of mixed language and its application

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

Anonymous

***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 μ C/OS-II kernel.**

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

I. Introduction

As a complementary approach to testing, formal methods have been prove to be an efficient way to guarantee the correctness and soundness of software system. We adhere that 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. 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.

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 [] integrate the semantics of C and assembler to verify operating systems and hypervisors. However, it is not easy to check the soundness of their semantics and the pen-and-paper proof is non-trivial. A checkable semantics with an automated toolkit for system analysis is expected.

$\mathbb{K}$ 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, deductive program verifier and a cross language program equivalence checker. $\mathbb{K}$ has been successfully applied in many industrial languages, such as C, java and javasript. In particular, C semantics in $\mathbb{K}$ passed 99.2% of the test cases for GCC, and has been commercialized by companies to verify the correctness and safety of C programs.

In this paper, we are focusing on achieving the goal of formally analyzing systems implemented in mixed language. 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 sum program implemented in mixed languages;
- formally analyzing the schedule mechanism of OSEK/VDX based automobile operating system with our semantics.

The rest of this paper is organized as follows. Section 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 C

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.

Configuration.

configuration=($Pro_{asm} :: Instr,\ Mem :: N \mapsto B^8,\ GReg :: R \mapsto B^8,\ SReg :: R \mapsto B^8,\ Stack :: List$)

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.

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.

$$< (\textbf{LDR}\ R_i,\ N),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \varepsilon,\ Mem,\ GReg[N/value(R_i)],\ SReg,\ Stack >$$

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.

$$< (\textbf{STR}\ R_i,\ A_i),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \varepsilon,\ Mem[value(R_i)/value(A_i)],\ GReg,\ SReg,\ Stack >$$

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 integer N into the stack.

$$< (\textbf{STMDB}\ sp,\ N),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \varepsilon,\ Mem,\ GReg,\ SReg,\ [N \cdot Stack/Stack] >$$

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.

$$< (\textbf{STMIB}\ R_i,\ \{R_j–R_n\}),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< STMIB\ [R_i] + 4,\ \{R_{j+1} - R_n\},\ Mem[value(R_j)/value(R_i)],\ GReg,\ SReg,\ Stack >$$

The instruction **LDMIA** pops the value from the stack and

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

Fig. 1. The syntax of the ARM assembly language

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.

$$< (\textbf{LDMIA}\ sp,\ R_i),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \ \varepsilon,\ Mem,\ GReg[N/value(R_i)],\ SReg,\ [N \cdot StackList/StackList] >$$

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.

$$< (\textbf{ADD}\ R_i,\ R_{j1},\ R_{j2}),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \ \varepsilon,\ Mem,\ GReg[(value(R_{j1}) + value(R_{j2}))/value(R_i)],\ SReg,\ Stack >$$

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.

$$< (\textbf{MRS}\ R_i,\ CPSR),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \varepsilon,\ Mem,\ GReg[value(CPSR)/value(R_i)],\ SReg,\ Stack >$$

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.

$$< (\textbf{MSR}\ CPSR,\ R_i),\ Mem,\ GReg,\ SReg,\ Stack > \ \rightarrow$$

$$< \varepsilon,\ Mem,\ GReg,\ SReg[value(R_i)/value(CPSR)],\ Stack >$$

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.

$$< (\textbf{BL}\ X,\ Mem,\ GReg,\ SReg,\ Stack,\ fun > \ \rightarrow$$

$$< \pi_2.fun(X),\ Mem,\ GReg,\ SReg,\ Stack,\ fun >$$

B. Semantics for IMP with procedure 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.

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.

Configuration.
$configuration = (Proc_c :: Stmts,\ Mem :: N \mapsto B,\ Env :: Id \mapsto N,\ Fun :: Id \mapsto (Paras,\ stmts),\ FStack :: List)$

Semantics

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 $Mem(Env(e_1))$.

$< e_1 * e_2,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< Mem(Env(e_1)) * Mem(Env(e_2)),\ Mem,\ Env,\ Fun,\ Fstack >$

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 $Mem(Env(X))$.

$< X : Id,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< Mem(Env(X)),\ Mem,\ Env,\ Fun,\ Fstack >$

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.

$< X = n,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

legacy $< \varepsilon,\ Mem[n/Mem(Env(X))],\ Env,\ Fun,\ Fstack >$

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.

(i)$< if\ e_1\ then\ e_2\ else\ e_3,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< e_2,\ Mem,\ Env,\ Fun,\ Fstack > \ \ if\ Mem(Env(e_1)) == true$

(ii)$< if\ e_1\ then\ e_2\ else\ e_3,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< e_3,\ Mem,\ Env,\ Fun,\ Fstack > \ \ if\ Mem(Env(e_1)) == false$

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.

(i)$< while\ e_1\ e_2,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< e_2;\ while\ e_1\ e_2,\ Mem,\ Env,\ Fun,\ Fstack > \ \ if\ Mem(Env(e_1)) == true$

(ii)$< while\ e_1\ e_2,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< \varepsilon,\ Mem,\ Env,\ Fun,\ Fstack > \ \ if\ Mem(Env(e_1)) == false$

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.

$< e(E),\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$< \pi_2.fun(X)[E/\pi_1.Fun(e)],\ Mem,\ \varepsilon,\ Fun,\ [(C.Proc_c,\ Env) \cdot Fstack)/Fstack] >$

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.

$< return\ e,\ Mem,\ Env,\ Fun,\ Fstack > \ \rightarrow$

$stmt$	::=	$Id = e \mid$ `if` $e_1\ e_2$ `else` $e_3 \mid$ `while` $e_1\ e_2 \mid FunctionCall \mid$ `return` e
e	::=	$Int \mid Id \mid e_1 * e_2 \mid e_1 + e_2 \mid e_1 \leq e_2 \mid !e \mid e_1 \wedge e_2$
$FunctionCall$	::=	$Id(e)$

$< Mem(Env(e)) \circ s_1,\ Mem,\ Env_1,\ Fun,\ [stackList/(S_1,\ Env_1) \cdot stackList] >$

III. Integrate C and ARM Semantics

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.

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.

Configuration

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

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

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

$\pi_1.con_{asm} =_{df} Greg \qquad \pi_2.con_{asm} =_{df} Sreg \qquad \pi_3.con_{asm} =_{df} stack$

$\pi_1.con_c =_{df} env \qquad \pi_2.con_c =_{df} fstack \qquad \pi_3.con_c =_{df} fun$

Semantics

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.

$< Pro_c = e(E),\ Mem,\ Con_{asm},\ Con_c > \quad \rightarrow$

$< \pi_2.\pi_3.con_c(e)[E/value\{R_0 - R_3\}],\ Mem,\ Con_{asm},\ Con_c[(next.Pro_c,\ \pi_1.Con_c) \cdot \pi_2.Con_c/\pi_2.Con_c,\ \varepsilon/\pi_1.Con_c] >$

$if\ |E| \leq 3$

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.

$< Pro_{asm} = B\ e,\ Mem,\ Con_{asm},\ Con_c > \quad \rightarrow$

$< \pi_2.\pi_3.Con_c(e)[valueR_0 - R_3/\pi_1.\pi_3.Con_c(e)],\ Mem,\ Con_{asm},\ Con_c >$

$if\ |\pi_1.\pi_3.Con_c(e)| \leq 3$

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.

$< Pro_{asm} = BL\ e,\ Mem,\ Con_{asm},\ Con_c > \quad \rightarrow$

$< \pi_2.\pi_3.Con_c(e)[valueR_0 - R_3/\pi_1.\pi_3.Con_c(e)],\ Mem,\ Con_{asm},\ Con_c >$

$if\ |\pi_1.\pi_3.Con_c(e)| \leq 3$

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.

$$< Pro_c = return\ e,\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow$$

$$< \varepsilon,\ Mem,\ Con_{asm}[\pi_1.Con_{asm}[e/R_0]/\pi_1.Con_{asm}],\ Con_c >$$

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.

$$< Pro_{asm} = END,\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow$$

$$< stmt_1,\ Mem,\ Con_{asm},\ Con_c[\pi_2.Con'_c/((stmt_1,\ env_1)\cdot\pi_2.Con'_c),\ env_1/\pi_1.Con_c] >$$

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.

$$< Pro_{asm} = (OP\ R_i,\ R_j,\ A_i),\ Mem,\ Con_{asm},\ Con_c > \ \rightarrow$$

$$< (OP\ R_i,\ R_j,\ Mem(A_i)),\ Mem,\ Con_{asm},\ Con_c >$$

IV. Simulation and Application of the Integrated Semantics

A. Implement the Semantics in K framework

The $\mathbb{K}$ Framework is a rewriting based executable semantic framework for defining programming languages, type systems and formal analysis tools using configurations and rules. We use $\mathbb{K}$ extensively in our work. Configurations in $\mathbb{K}$ organize the state in units called cells, which are labeled and can be nested. *Rules* in $\mathbb{K}$ rewrite *configurations* to define valid state transitions. $\mathbb{K}$'s foundations are based in Reachability Logic, which has a sound and relatively complete proof system for reasoning about transition systems.

We give practical insights into the $\mathbb{K}$ framework by defining the integer subset of the ARM assembly language. Whereas we defined the entire subset of integer-based instructions, for brevity we only describe a representative snippet of it. Apart from the instruction set presented in Section II, a number of pseudo-instructions appear in the executables that we analyze; we have also defined those in $\mathbb{K}$, but we also omit them here. The general methodology for language definitions in $\mathbb{K}$ begins with the (abstract) syntax, determines the configuration, and then gives the semantic rules.

The syntax of the integrated language is the union of IMP and assembly, which we omit the introduction here. 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 $\mathbb{K}$ cells hold the necessary semantic infrastructure (registers, instruction cache, memory, etc.). Two cells appear in most $\mathbb{K}$ definitions: a cell whose label is $\top$ that encloses all the other cells, and a cell labeled k that holds the computation.

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. 2. Semantic rules for integrated language

We implement the semantics of integrated language in $\mathbb{K}$ as shown in Figure 2, 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.

B. 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;}
```

```
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

We looked into the implementation of an industrial OSEK-based operating system (OSEK OS for abbreviation) kernel, whose formal semantics of system calls have been defined in []. 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 [] 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 in figure **??**

```
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();
}
```

VI. Related Work

A. integrate semantics

B. verification of systems implemented in mixed language

VII. Discussion and Conclusion

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.