# Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT*

Sabine Schmaltz and Andrey Shadrin

Saarland University, Germany
{sabine,shavez}(at)wjpserver.cs.uni-saarland.de

**Abstract.** Pervasive formal verification of operating systems and hypervisors is, due to their safety-critical aspects, a highly relevant area of research. Many implementations consist of both assembler and C functions. Formal verification of their correctness must consider the correct interaction of code written in these languages, which is, in practice, ensured by using matching application binary interfaces (ABIs). Also, these programs must be able to interact with hardware. We present an integrated operational small-step semantics model of intermediate-language C and *Macro-Assembler* code execution for pervasive operating systems and hypervisor verification. Our semantics is based on a compiler calling convention that defines callee- and caller-save registers. We sketch a theory connecting this semantic layer with an ISA-model executing the compiled code for use in a pervasive verification context. This forms a basis for soundness proofs of tools used in the VerisoftXT project and is a crucial step towards arguing formal correctness of execution of the verified code on a gate-level hardware model.

## 1 Introduction

For operating systems, correctness of implementation is highly desirable – in particular for safety-critical embedded systems such as cars or airplanes. Hypervisors are employed to partition system resources efficiently, providing strictly-separated execution contexts in which operating systems can again be run. To argue implementation correctness of a system consisting of both a hypervisor and operating systems, formal verification can provide solid evidence if done in a pervasive way.

The L4verified kernel [1] is an example of a recent operating system verification effort that has achieved impressive code verification results. A detailed memory model for low-level pointer programs in C was applied in combination with separation logic [2]. The assembler portions have not been verified in conjunction with the C code yet to our knowledge. Judging from their choice of C semantics, however, we are certain that all gaps present can be closed with minimal additional effort when the right models and theories are applied. In this paper, we present a description of such a theory for C and assembler code verification.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

The FLINT group, on the other hand focuses on assembler code verification using their framework XCAP [3], which they successfully applied in [4] and [5]. So far, however, no integration of results into a semantics stack with high-level programming languages has been reported yet.

In a pervasive verification effort that aims at code verification above assembler-level, compiler correctness is crucial. During the Verisoft project, a compiler for the C-like language C0 was verified [6]. A mildly optimizing compiler that translates C-minor (a subset of C) to PowerPC assembly code has been verified by Xavier Leroy et al. [7, 8] and used in a pervasive verification effort by Andrew Appel [9]. Both these efforts made use of interactive theorem provers.

In the scope of this paper we provide descriptions of the *Macro-Assembler* (short: *MASM*, section 2) and the C intermediate-language (*C-IL*, section 3) semantics we use to construct the integrated *C-IL+MASM*-semantics we propose in section 4. We combine a rather high-level assembler-semantics with a low-level C-intermediate-language semantics. This results in a model in which function calls between the two languages have the straightforward semantics we expect according to compiler calling conventions. We describe how we apply pervasive theory in section 5 to prove that our inter-language-call-semantics is sound with respect to the underlying machine-code execution model. Note that all of this can still be considered work-in-progress since none of the proofs have been checked in a theorem prover.

As main contributions of this paper, we consider, first, our unconventional choice to design both *C*- and assembler-semantics in such a way that they can interoperate easily – resulting in a model that accurately captures the compiler calling convention –, and, second, our demonstration that such an integrated model can be easily justified using pervasive compiler correctness theory.

## 2 Macro-Assembler Semantics

One might wonder why, in a pervasive verification effort, there is any need for a high-level assembler semantics. Operating systems and hypervisors implemented in C and assembler are generally compiled to machine code – a machine-code execution model technically is fully sufficient to argue about such systems. However, doing this, we would discard all the comfort and gain of speed that appropriate abstraction can provide. We consider a machine-code execution model that we refer to using the name *ISA-Assembler*. It is characterized by the following: Instructions are executed as atomic transitions – an instruction pointer register points to the next instruction in memory.

Concerning code verification, the *ISA-Assembler*-model has one particular drawback: It provides no useful abstraction in terms of control flow. While this is true for the language of machine code, the assembler code used in operating systems has structure we can exploit during formal verification: Our assembler code is called from or calls functions of a stack-based programming language. Instead of executing machine code instructions from the configuration's memory at the instruction pointer, we apply abstraction techniques normally used in high-level programming language semantics definitions. We gain an easy-to-understand model of high-level assembler code execution that can be integrated with a C model in a straightforward way. This comes at a cost: In order to obtain a more simple model of assembler code execution, we enforce a certain structure of code which may exclude well-behaving assembler programs. However, all code we want to verify has this structure.

We introduce *Macro-Assembler* (*MASM*) as a restricted assembler language: All targets of branch or jump instructions are either local labels or names of functions – we model the control flow of *Macro-Assembler* as a labeled transition system. In order to make integration of *MASM* with a stack-based programming language very simple, we abstract from the concrete stack layout in memory by introducing a stack component in form of a list of abstract stack frames to the configuration.

In [10], a stack-based typed low assembly language is proposed as a target language for code verification. The authors can encode any compiler calling conventions in their type system since everything about the stack including the stack frame header layout is exposed. In our work we abstract the stack away hiding all details that are compiler relevant. Additionally, we provide a feature found in some assembler languages: uses lists that specify the registers used by an assembler function. The *MASM*-compiler inserts instructions that save/restore these registers in the prologue/epilogue of the compiled assembler function. In the following, we present formal definitions to elaborate on the structure of *MASM*.

**Configuration** A *Macro-Assembler* configuration

$$c = (c.\mathcal{M}, c.\mathit{regs}, c.s) \in \mathit{conf}_{\mathit{MASM}}$$

consists of a byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}^{8}$ (where $k$ is the number of bytes in a machine word and $\mathbb{B} \equiv \{0, 1\}$), a component $\mathit{regs} : \mathcal{R} \to \mathbb{B}^{8k}$ that maps register names to their values, and an abstract stack $s : \mathit{frame}^{*}_{\mathit{MASM}}$. Each frame

$$s[i] = (p, \mathit{loc}, \mathit{saved}, \mathit{pars}, \mathit{lifo})$$

contains the name $p$ of the assembler function we are executing in, the location *loc* of the next instruction to be executed in $p$'s body, a component *saved* that is used to store values of callee-save registers used by the function, a component *pars* that represents the parameter region of the stack frame, and a component *lifo* that represents the part of the stack where data can be $push$ed and $pop$ped to/from.

**Program** A *Macro-Assembler* program $\pi$ is a procedure table that maps function names $p$ to procedure table entries:

$$\pi(p) = (\mathit{npar}, \mathcal{P}, \mathit{uses})$$

Here, *npar* describes the total number of machine word parameters of the function, $\mathcal{P} : \mathit{instr}^{*}_{\mathit{MASM}}$ is a list of *Macro-Assembler* instructions representing the procedure body, and *uses*: $\mathcal{R}^{*}$ is a list of register names to be saved and restored.

In case the following instructions are not provided by the underlying hardware, we implement them as assembler macros: $\mathit{call}, \mathit{ret}, \mathit{push}, \mathit{pop}$. An assembler macro is simply a shorthand for a sequence of assembler instructions. *MASM* can easily be extended by the notion of user-defined macros, however, we have not done so yet.

$$\frac{\mathit{instr}_{\mathit{next}}(c) = \textbf{instr}(i) \qquad \mathit{MASM\text{-}to\text{-}ISA}(c) \underset{\mathit{ISA}}{\rightarrow} d'}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{ISA\text{-}to\text{-}MASM}(d')} \ (\textsc{Instr}) \qquad \frac{\mathit{instr}_{\mathit{next}}(c) = \textbf{goto } l}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)} \ (\textsc{Goto})$$

$$\frac{\mathit{instr}_{\mathit{next}}(c) = \textbf{ifnez } r \textbf{ goto } l \qquad c.\mathit{regs}(r) = 0^{8k}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{inc}_{\mathit{loc}}(c)} \ (\textsc{Goto-Fail}) \qquad \frac{\mathit{instr}_{\mathit{next}}(c) = \textbf{ifnez } r \textbf{ goto } l \qquad c.\mathit{regs}(r) \neq 0^{8k}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)} \ (\textsc{Goto-Succ})$$

$$\frac{\begin{array}{c}\mathit{instr}_{\mathit{next}}(c) = \textbf{push } r \\ \textbf{hd}(c.s) = (p, \mathit{loc}, \mathit{saved}, \mathit{pars}, \mathit{lifo})\end{array}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{lifo}}(\mathit{inc}_{\mathit{loc}}(c), c.\mathit{regs}(r) \circ \mathit{lifo})} \ (\textsc{Push}) \qquad \frac{\begin{array}{cc}\mathit{instr}_{\mathit{next}}(c) = \textbf{call } p & \pi(p).\mathit{npar} - 4 \leq \textbf{hd}(c.s).\mathit{lifo} \\ \mathit{call}_{\mathit{frame}}(c, p, \mathit{frame}_{\mathit{new}}) & c' = \mathit{drop}_{\mathit{lifo}}(c, \pi(p).\mathit{npar} - 4)\end{array}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} c'[s := \mathit{frame}_{\mathit{new}} \circ \mathit{inc}_{\mathit{loc}}(c').s]} \ (\textsc{Call})$$

$$\frac{\mathit{instr}_{\mathit{next}}(c) = \textbf{pop } r \qquad \mathit{lifo} \neq [] \qquad \textbf{hd}(c.s) = (p, \mathit{loc}, \mathit{saved}, \mathit{pars}, \mathit{lifo})}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{set}_{\mathit{lifo}}(\mathit{set}_{\mathit{reg}}(c, r, \textbf{hd}(\mathit{lifo})), \textbf{tl}(\mathit{lifo}))} \ (\textsc{Pop}) \qquad \frac{\mathit{instr}_{\mathit{next}}(c) = \textbf{ret}}{\pi \vdash c \underset{\mathit{MASM}}{\rightarrow} \mathit{drop}_{\mathit{frame}}(\mathit{restore}_{\mathit{saved}}(c))} \ (\textsc{Ret})$$

**Table 1.** Operational semantics of *MASM*

**Semantics** Since *Macro-Assembler* is compiled to machine code, *Macro-Assembler* semantics for basic instructions can be inferred from the semantics of *ISA-Assembler*. The main differences stem from the distinct modeling of control-flow: In *ISA-Assembler*, we have a global instruction pointer whereas in *Macro-Assembler* we have local program location and stack-abstraction. Whenever a *Macro-Assembler* instruction accesses the stack-region, instead of updating/reading the memory, we update/read the corresponding component of the abstract stack. In case it is not trivially possible to find an equivalent update on the abstract stack, we consider the program in question illegal – or unsuitable for use with our semantics. This concerns, in particular, all instructions that explicitly update the stack pointer registers (this would break our stack abstraction) or write to addresses of the physical stack that describe return addresses or previous frame base addresses.

In the *call*-rule presented in table 1, $\mathit{call}_{\mathit{frame}}(c, p, \mathit{frame}_{\mathit{new}})$ is a predicate over a configuration $c$, a function $p$, and a frame $f$ that enforces following conditions:

$$\mathit{frame}_{\mathit{new}}.p = p, \quad \mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \quad \mathit{frame}_{\mathit{new}}.\mathit{saved} = c.\mathit{regs}|_{\pi(p).\mathit{uses}}$$

$$\mathit{frame}_{\mathit{new}}.\mathit{pars}[\pi(p).\mathit{npar} - 1 : 4] = \mathit{read}_{\mathit{lifo}}(c, \pi(p).\mathit{npar} - 4), \quad \mathit{frame}_{\mathit{new}}.\mathit{lifo} = []$$

All registers in the uses list of the function are saved in the new frame, and parameters that are passed on the stack are moved from the *lifo*-component of the top-most frame to the *pars*-component of the new frame. The calling convention we assume states that the first four parameters are passed in registers (space on the stack is reserved nonetheless), while the remaining parameters are passed on the stack (in right-to-left order).

## 3 C Semantics

As countless others have noted before [11, 12], there is not "the" C semantics: the term "C" describes an equivalence class of semantics that fall under the scope of what is commonly called the programming language C and its standard library. Depending on architecture and compiler, semantics may differ for the underspecified areas.

|  |  |
|---|---|
| $t \in \mathbb{T}$ | $t \in \mathbb{T}_P$ |
| **struct** $t_C \in \mathbb{T}$ | $t_C \in \mathbb{T}_C$ |
| **array**$(t, n) \in \mathbb{T}_{\textbf{ptr}}$ | $t \in \mathbb{T}, n \in \mathbb{N}$ |
| **ptr**$(t) \in \mathbb{T}_{\textbf{ptr}}$ | $t \in \mathbb{T}$ |
| **fptr**$(t, T) \in \mathbb{T}_{\textbf{ptr}}$ | $t \in \mathbb{T}, T \in \mathbb{T}^*$ |

**Table 2.** The set $\mathbb{T}$ of types of *C-IL*

|  |  |
|---|---|
| **val**$(b, \mathbf{i}i) \in val$ | $b \in \mathbb{B}^i$ |
| **val**$(b, \mathbf{u}i) \in val$ | $b \in \mathbb{B}^i$ |
| **val**$(B, \textbf{struct } t_C) \in val_{\textbf{struct}}$ | $B \in (\mathbb{B}^8)^*$ |
| **val**$(b, t) \in val_{\textbf{ptr}}$ | $b \in \mathbb{B}^{size_{ptr}}, t \in \mathbb{T}_{\textbf{ptr}}$ |
| **lref**$((v, o), i, t) \in val_{\textbf{lref}}$ | $v \in \mathbb{V}, o, i \in \mathbb{N}, t \in \mathbb{T}_{\textbf{ptr}}$ |
| **fun**$(f) \in val_{\textbf{fun}}$ | $f \in \mathbb{F}_{name}$ |

**Table 3.** The set $val$ of values of *C-IL*

Since C is a programming language with an overwhelming complexity – much of which is redundant –, we consider an intermediate language for C that we call *C-IL*. Note that, instead of defining Pascal with C syntax (as has been done in Verisoft), we now consider a semantics that really captures the low-level features of C. We do not consider side-effects in expressions, and neither do we put much effort on modeling C syntax. These, we leave to the layer above, where we can define C based on *C-IL*.

The intermediate language we consider has been designed with some very specific features – optimized for integration with *Macro-Assembler*. Like *MASM*, *C-IL* is a goto-language defined in the form of a labeled transition system.

Since we want to do lowest-level operating systems verification, we only consider a global byte-addressable memory and an abstract stack. We do not consider the heap as a separate memory since the notion of a heap only exists when there is some form of memory allocation system available (e.g. the one provided by the standard library or the operating system). Pointer arithmetics is allowed on pointers to the global memory to the full extent possible. For local variables we restrict pointer arithmetics to calculating offsets inside local variable memories. In the semantics we propose, every memory access corresponds to dereferencing a left value – a left value is either a pointer to the global memory or a reference to some offset in a local variable.

In interrupt descriptor tables, we need to store function pointer values, thus, we explicitly model the addresses of functions. Obviously, these cannot be derived from C semantics since they depend only on where in memory the program resides, thus we give them as parameter to the semantics.

One main issue of C is its dependency on the underlying architecture and compiler. We suggest that semantics for C should be parameterizable to make it applicable to at least the most common cases.

In the following, $\mathbb{V}$ is a set of variable names, $\mathbb{F}$ is a set of field names, $\mathbb{F}_{name}$ is a set of function names. We use the notation $\mathcal{X}^* \equiv \bigcup_{n=1}^{\infty} \mathcal{X}^n \cup \{[]\}$ to describe the set of lists/strings with elements from the set $\mathcal{X}$. A list of length $n$ with elements from $\mathcal{X}$ is given by $x = (x_{n-1}, \ldots, x_0) = x[n-1 : 0] \in \mathcal{X}^n$ and we define the shorthand $x[i] \overset{def}{=} x_i$.

**Types** For every instance of *C-IL*, we assume a set of primitive types $\mathbb{T}_P$ to be given such that $\mathbb{T}_P \subset \{\textbf{void}\} \cup \bigcup_{i=0}^{\infty} \{\mathbf{i}i, \mathbf{u}i\}$ describes a set of basic signed (**i**) and unsigned (**u**) integer types of size $i$ (given in bits). Usually, we consider sizes which are multiples of 8. Further, we assume a set $\mathbb{T}_C$ of struct type names to be given.

We define the set of types $\mathbb{T}$ inductively in table 2. Array types are given by their element type and length. Function pointer types are identified by their return value type and a list of their parameters' types. Note that struct types are always identified by the corresponding composite type name $t_C$. The actual type definition of a struct type can be found by looking it up in the program's struct type declaration (defined later).

**Values** We represent most values using bit-strings or byte-strings. This is owed to the fact that *C-IL* is designed for use in conjunction with hardware models. For struct types, we consider byte-strings as their value (see table 3). We only need them in order to model struct assignment since access to a field of a struct is performed by calculating the corresponding left value followed by a precise memory access. A pointer to the global memory is a value $\mathbf{val}(b, t)$ consisting of an address and a pointer type.

Due to the stack abstraction used, we have to treat pointers to local variables differently: We represent these *local references* $\mathbf{lref}((v, o), i, t)$ by variable name $v$ and offset $o$ inside that variable. Additionally, we have the number $i$ of the stack frame the local reference refers to and a pointer type $t$.

For functions $f$ we do not need the exact address of (since we do not need to store their function pointers in memory), we introduce a symbolic function value $\mathbf{fun}(f)$.

**Expressions** We define the set of expressions $\mathbb{E}$ in table 4. $\mathbb{O}_1$ and $\mathbb{O}_2$ are sets of operators (table 5) defined for the compiler in question. A unary operator is a partial function $\oplus : val \rightharpoonup val$, whereas a binary operator is a function $\oplus : val \times val \rightharpoonup val$. Operators are provided for each type they are meaningful for. All expressions are strictly typed in *C-IL* – when translating from *C* to *C-IL*, type casts need to be inserted explicitly.

**Statements** *C-IL* uses a reduced set of statements (see table 6) consisting of assignment, goto, if-not-goto, function call, procedure call, and corresponding return statements. Goto statements specify the target destination in form of a label (the index of the target statement in the program).

**Configuration** A *C-IL* configuration

$$c = (\mathcal{M}, s) \in \mathit{conf}_{\text{C-IL}}$$

consists of a global, byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}^8$ and a stack $s \in \mathit{frame}^*_{\text{C-IL}}$ which is a list of *C-IL*-frames. A *C-IL*-frame

$$s[i] = (\mathcal{M}_\mathcal{E}, \mathit{rds}, f, \mathit{loc}) \in \mathit{frame}_{\text{C-IL}}$$

consists of a local memory component $\mathcal{M}_\mathcal{E} : \mathbb{V} \to (\mathbb{B}^8)^*$ which maps variable names to local byte-offset-addressable memories (represented as lists of bytes), a return destination component $rds : val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}$ which is either a pointer to where the return value of the function is going to be stored when it returns or the value $\bot$ denoting the absence of a return destination, a function name $f$ which describes the function we are executing and a location $loc \in \mathbb{N}$ which describes where in $f$'s body execution should continue.

| | | |
|---|---|---|
| constants | $c$ | $c \in val$ |
| variable names | $v$ | $v \in \mathbb{V}$ |
| function names | $f$ | $f \in \mathbb{F}_{name}$ |
| unary operation | $\oplus e$ | $e \in \mathbb{E}$ and $\oplus \in \mathbb{O}_1$ |
| binary operation | $(e_1 \oplus e_2)$ | $e_1, e_2 \in \mathbb{E}$ and $\oplus \in \mathbb{O}_2$ |
| ternary operation | $(e \,?\, e_1 : e_2)$ | $e, e_1, e_2 \in \mathbb{E}$ |
| type cast | $(t)e$ | $t \in \mathbb{T}$ and $e \in \mathbb{E}$ |
| dereferencing | $*e$ | $e \in \mathbb{E}$ |
| address-of | $\&e$ | $e \in \mathbb{E}$ |
| field access | $(e).f$ | $e \in \mathbb{E}$ and $f \in \mathbb{F}$ |
| size of type | **sizeof**$(t)$ | $t \in \mathbb{T}$ |
| size of expression | **sizeof**$(e)$ | $e \in \mathbb{E}$ |

**Table 4.** The set $\mathbb{E}$ of *C-IL*-expressions

| | |
|---|---|
| unary operators | $\mathbb{O}_1 = \{-, \sim, !\}$ |
| binary operators | $\mathbb{O}_2 = \{+, -, *, /, \%, <<, >>, <, >, <=, >=, ==, !=, \&, \mid, \hat{}, \&\&, \mid\mid\}$ |

**Table 5.** Operators of *C-IL*

| | |
|---|---|
| $e_0 = e_1$ | $e_0, e_1 \in \mathbb{E}$ |
| **goto** $l$ | $l \in \mathbb{N}$ |
| **ifnot** $e$ **goto** $l$ | $e \in \mathbb{E}, l \in \mathbb{N}$ |
| $e_0 =$ **call** $e(E)$ | $e_0, e \in \mathbb{E}, E \in \mathbb{E}^*$ |
| **call** $e(E)$ | $e \in \mathbb{E}, E \in \mathbb{E}^*$ |
| **return**, **return** $e$ | $e \in \mathbb{E}$ |

**Table 6.** The set $\mathbb{S}$ of *C-IL*-statements

**Program** A *C-IL* program

$$\pi = (\mathcal{F}, \mathcal{V}_G, T_F)$$

consists of a function table $\mathcal{F}$, a declaration of global variables $\mathcal{V}_G : (\mathbb{V} \times \mathbb{T})^*$ consisting of pairs of variable names and types, and a struct type declaration $T_F : \mathbb{T}_C \to (\mathbb{F} \times \mathbb{T})^*$ which returns for every composite type name a declaration of its fields.

A function table entry

$$\pi.\mathcal{F}(f) = (npar, \mathcal{V}, \mathcal{P})$$

contains the number of parameters $npar$ of the function $f$, a local variable and parameter declaration $\mathcal{V} : (\mathbb{V} \times \mathbb{T})^*$ and a function body $\mathcal{P} : \mathbb{S}^*$.

**Context** Configuration and program are not enough: we need additional information in order to execute a *C-IL* program. For this, we introduce a context $\theta$ which provides all missing information. It contains information on the endianness of the underlying architecture (i.e. byte-order used), the addresses of global variables in memory, function pointer addresses (given by a partial, injective function $\theta.\mathcal{F}_{adr}$), offsets of fields in struct types, sizes of struct types, a type-casting function that matches the behavior of the compiler, and the type used by the compiler for results of the `sizeof`-operator.

**Expression Evaluation** Expressions are evaluated by a function that returns either a *C-IL*-value or the special value $\perp$ that denotes that the expression cannot be evaluated:

$$[e]_c^{\pi,\theta} \in val \cup \{\perp\}$$

Depending on the expression, we may need the complete state, i.e. configuration, program and context, to evaluate it. Since expression evaluation is defined in the obvious way, given the choices we made, we omit its definition to save space.

$$\frac{\mathit{stmt}_{\mathit{next}}(c) = e_0 = e_1}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{inc}_{\mathit{loc}}(\mathit{write}^{\theta}(c, [\&e_0]_c^{\theta,\pi}, [e_1]_c^{\theta,\pi}))} \text{ (ASSIGN)} \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{goto } l}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)} \text{ (GOTO)}$$

$$\frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{ifnot } e \textbf{ goto } l \qquad \textbf{zero}([e]_c^{\theta,\pi})}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{set}_{\mathit{loc}}(c, l)} \text{ (IFNOTGOTO-SUCC)} \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{ifnot } e \textbf{ goto } l \qquad \neg\textbf{zero}([e]_c^{\theta,\pi})}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{inc}_{\mathit{loc}}(c)} \text{ (IFNOTGOTO-FAIL)}$$

$$\frac{\begin{array}{c}\mathit{stmt}_{\mathit{next}}(c) = \textbf{call } e(E) \vee \mathit{stmt}_{\mathit{next}}(c) = e_0 = \textbf{call } e(E) \\ \mathit{is\text{-}function}([e]_c^{\theta,\pi}, f) \qquad \mathit{call}_{\mathit{frame}}(c, f, E, \mathit{frame}_{\mathit{new}})\end{array}}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} c[s := \mathit{frame}_{\mathit{new}} \circ \mathit{inc}_{\mathit{loc}}(c).s]} \text{ (CALL)} \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{return}}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{drop}_{\mathit{frame}}(c)} \text{ (RETURN)}$$

$$\frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{return } e \qquad c.\mathit{rds}_{\mathit{top}} \neq \bot}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{write}^{\theta}(\mathit{drop}_{\mathit{frame}}(c), c.\mathit{rds}_{\mathit{top}}, [e]_c^{\theta,\pi})} \text{ (RETURNVAL1)} \qquad \frac{\mathit{stmt}_{\mathit{next}}(c) = \textbf{return } e \qquad c.\mathit{rds}_{\mathit{top}} = \bot}{\pi, \theta \vdash c \underset{\mathit{C\text{-}IL}}{\rightarrow} \mathit{drop}_{\mathit{frame}}(c)} \text{ (RETURNVAL2)}$$

**Table 7.** Operational semantics of *C-IL*

**Memory Semantics** On the one hand we have byte-addressable memories, on the other we have typed values. We provide functions $\mathit{read}^{\theta} : \mathit{conf}_{\mathit{C\text{-}IL}} \times (\mathit{val}_{\textbf{ptr}} \cup \mathit{val}_{\textbf{lref}}) \to \mathit{val}$ and $\mathit{write}^{\theta} : \mathit{conf}_{\mathit{C\text{-}IL}} \times \mathit{val} \times (\mathit{val}_{\textbf{ptr}} \cup \mathit{val}_{\textbf{lref}}) \to \mathit{conf}_{\mathit{C\text{-}IL}}$ which, respectively, dereference a pointer value in a given configuration (read from memory) or write a given value to memory, resulting in a new configuration. To specify their effect, similar functions ($\mathit{read}^{\theta}_{\mathcal{E}}$, $\mathit{write}^{\theta}_{\mathcal{E}}$) are provided to read and write a local variable/parameter (identified by variable name) from a stack frame.

Note that, since we do not model addresses of local variables explicitly (this would either expose stack layout or require a more sophisticated memory), our semantics carries the limitation that pointers to local variables cannot be stored in memory.

**Operational Semantics** In table 7, we give operational semantics of *C-IL*. **zero** is a predicate that is true when the given *C-IL*-value is a representation of zero. The next statement to be executed in the given configuration is computed by $\mathit{stmt}_{\mathit{next}}$ from program/location of the top-most stack frame. With $\mathit{inc}_{\mathit{loc}}$ and $\mathit{drop}_{\mathit{frame}}$ we produce configurations in which, respectively, the location counter of the top-most frame is incremented or the top-most frame is simply removed from the stack.

In the *call*-rule, the new stack frame $\mathit{frame}_{\mathit{new}}$ is chosen nondeterministically according to the following constraints (represented by the predicate $\mathit{call}_{\mathit{frame}}(c, f, E, \mathit{frame}_{\mathit{new}})$):

$$\forall 0 \leq i < \mathit{npar} : \quad \mathit{read}^{\theta}_{\mathcal{E}}(\mathit{frame}_{\mathit{new}}, v_i, 0, t_i) = [E[i]]_c^{\theta,\pi}$$

$$\forall \mathit{npar} \leq i < \textbf{len}(\mathcal{V}) : \quad \textbf{len}(\mathit{frame}_{\mathit{new}}.\mathcal{M}_{\mathcal{E}}(v_i)) = \mathit{size}(t_i)$$

$$\mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \quad \mathit{frame}_{\mathit{new}}.f = f, \quad \mathit{frame}_{\mathit{new}}.\mathit{rds} = \begin{cases} [\&e_0]_c^{\theta,\pi} & \text{for function call} \\ \bot & \text{for procedure call} \end{cases}$$

Here, $(v_i, t_i) = \mathcal{V}[i]$ is the $i$-th declaration in function $f$'s parameters and local variable declaration $\mathcal{V} = \pi.\mathcal{F}(f).\mathcal{V}$, and $\mathit{npar} = \pi.\mathcal{F}(f).\mathit{npar}$ is the number of parameters of the function $f$. Note that we only place a strict constraint on the parameter values: initial content of local variables is chosen nondeterministically with appropriate size for the declared type.

## 4 Integrated *C-IL+MASM*-Semantics

We extend both *C-IL* and *MASM* in such a way that we can do the final step of integrating them into *C-IL+MASM*. To achieve this, we define a compiler calling convention and apply it to interface the two languages. The goal of this integration is to 'slice' the model stack horizontally, providing a self-contained model to argue about a system layer that involves both *C-IL* and *MASM* code execution.

In the first Verisoft project, whenever assembler code is encountered, the compiler simulation relation is applied to reach an equivalent *ISA-Assembler*-configuration from which to execute the assembler code. The proposed integrated semantics simply provides another layer of abstraction on top of such a model. The assembler verification approach that was used in the VerisoftXT project is based on translating assembler code to C so that it can be verified using a C verification tool [13]. We can benefit from the abstraction we introduce here in a soundness proof for the assembler verification approach: Instead of proving a simulation between *ISA-Assembler* and *C-IL* (which would require a substantial amount of software conditions), we can prove a simpler simulation between *MASM* and *C-IL*. In turn, we have to prove correct compilation for *MASM*.

There is a restriction on the interaction between *C-IL* and *MASM*: currently, we only allow primitive values to be passed between *C-IL*- and *MASM*-functions.

### 4.1 Calling Convention

The compiler calling convention describes the interface between the caller and the callee. In our experiments, we consider a compiler calling convention given by the following rules:

1. The first four parameters are passed through Registers $R_{p_1}, R_{p_2}, R_{p_3}, R_{p_4}$.
2. The remaining parameters (if existent) are passed on the stack in right-to-left order. There is space reserved on the stack for parameters passed in registers.
3. The return value is passed through register $R_{rv}$.
4. All callee-save registers (given by $\mathcal{R}_{\textbf{callee}} \subset \mathcal{R}$) must be restored before return.
5. The callee is responsible for cleaning up the stack.
6. $R_{p_1}, R_{p_2}, R_{p_3}, R_{p_4}, R_{rv} \notin \mathcal{R}_{\textbf{callee}}$.

### 4.2 Semantics

In order to obtain an integrated model of *C-IL* and *MASM*, there are two things left to do: define how we model the state of the combined semantics and define transitions.

Probably the most basic way to define a configuration of *C-IL+MASM* is to consider a list of alternating *C-IL*- and *MASM*-configurations that represents the call stack between the two languages. The top-most configuration we consider *active* while we consider the rest of them *inactive*. One 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 *C-IL* and *MASM*. An *execution context* is a configuration of the corresponding language where the memory component $\mathcal{M}$ is removed:

$$s \in \mathit{context}_{\mathit{C\text{-}IL}} \equiv \mathit{frame}^*_{\mathit{C\text{-}IL}}, \qquad (\mathit{regs}, s) \in \mathit{context}_{\mathit{MASM}} \equiv (\mathcal{R} \to \mathbb{B}^{8k}) \times \mathit{frame}^*_{\mathit{MASM}}$$

Another observation we make is that in order to integrate the two semantics, we need to add information to inactive *C-IL*-execution-contexts: It would be very nice to know where the *C-IL*-execution context will store the return value that is passed in $R_{rv}$ when it becomes active again. Another notion we want to capture in the semantics is that the *C-IL*-compiler may rely on the callee-save convention being respected by the programmer: When the callee-save registers have modified values, there is no guarantee whatsoever that execution of the *C-IL*-code will continue as expected. We define the inactive *C-IL*-execution-context

$$(\mathit{rds}, \mathit{regs}_{\mathbf{callee}}, s) \in \mathit{context}^{\mathit{inactive}}_{\mathit{C\text{-}IL}}$$

which consists of a return destination pointer $\mathit{rds} \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}$, a function $\mathit{regs}_{\mathbf{callee}} : \mathcal{R}_{\mathbf{callee}} \rightharpoonup \mathbb{B}^{8k}$ which describes the content of callee-save registers expected when control is returned to the execution context, and a *C-IL*-stack $s \in \mathit{frame}^*_{\mathit{C\text{-}IL}}$.

The last observation is that it is not meaningful to store register values in the inactive *MASM*-execution-context, with one exception: we can keep the values of callee-save registers, since, assuming the *C-IL*-compiler respects the calling conventions, they will be restored when control is returned to the context. We define the inactive *MASM*-execution-context

$$(\mathit{regs}_{\mathbf{callee}}, s) \in \mathit{context}^{\mathit{inactive}}_{\mathit{MASM}}$$

to consist of a callee-save register file $\mathit{regs}_{\mathbf{callee}} : \mathcal{R}_{\mathbf{callee}} \to \mathbb{B}^{8k}$ that holds the values of callee-save registers belonging to the execution context, and a *MASM*-stack $s \in \mathit{frame}^*_{\mathit{MASM}}$.

**Configuration** A *C-IL+MASM*-configuration

$$c = (\mathcal{M}, ac, sc) \in \mathit{conf}_{\mathit{C\text{-}IL+MASM}}$$

consists of the same byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}$ we have seen before, the active execution context $ac : \mathit{context}_{\mathit{C\text{-}IL}} \cup \mathit{context}_{\mathit{MASM}}$, and a list of inactive execution contexts $sc \in (\mathit{context}^{\mathit{inactive}}_{\mathit{C\text{-}IL}} \cup \mathit{context}^{\mathit{inactive}}_{\mathit{MASM}})^*$.

**Program & Context** A *C-IL+MASM*-program $\pi = (\pi_{\mathit{C\text{-}IL}}, \pi_{\mathit{MASM}})$ is simply a pair of a *C-IL*-program and a *MASM*-program. Since we need context information about the compiler to execute *C-IL*, we keep the context $\theta$ from *C-IL*.

**Transitions** Essentially, we have three types of steps: We perform a pure *C-IL*- or *MASM*-step, an external function from the other language is called, or we return to the other language. Considering a given configuration, it is easy to decide which of these has to happen next: Calling a function which is not declared in the current language's program must be an external call. Executing a return-statement or -instruction when the stack of the active context is empty should return to the newest context from the list of inactive contexts. Everything else is a pure step. Let $\mathit{ext}(c)$ denote a predicate that checks in the described way whether the next step is an external step. Table 8 shows inference rules that describe *C-IL+MASM*'s transitions.

$$\frac{c.ac = s \qquad \neg ext(c) \qquad \pi.\pi_{\mathit{C\text{-}IL}}, \theta \vdash (c.\mathcal{M}, s) \to_{\mathit{C\text{-}IL}} c'_{\mathit{C\text{-}IL}}}{\pi, \theta \vdash c \to_{\mathit{C\text{-}IL+MASM}} c\,[\,\mathcal{M} := c'_{\mathit{C\text{-}IL}}.\mathcal{M},\ ac := c'_{\mathit{C\text{-}IL}}.s\,]} \qquad \text{(PURE-C-IL)}$$

$$\frac{c.ac = (\mathit{regs}, s) \qquad \neg ext(c) \qquad \pi.\pi_{\mathit{MASM}} \vdash (c.\mathcal{M}, \mathit{regs}, s) \to_{\mathit{MASM}} c'_{\mathit{MASM}}}{\pi, \theta \vdash c \to_{\mathit{C\text{-}IL+MASM}} c\,[\,\mathcal{M} := c'_{\mathit{MASM}}.\mathcal{M},\ ac := (c'_{\mathit{MASM}}.\mathit{regs}, c'_{\mathit{MASM}}.s)\,]} \qquad \text{(PURE-MASM)}$$

$$\frac{\begin{array}{c} c.ac = s \qquad ext(c) \qquad \mathit{stmt}_{\mathit{next}}(s, \pi.\pi_{\mathit{C\text{-}IL}}) = e_0 = \textbf{call}\ e(E) \\ \mathit{is\text{-}function}([e]_c^{\theta,\pi}, f) \qquad \mathit{CIL2MASM}_{\mathit{ctxt}}(c, f, E, \mathit{regs}_{\textbf{callee}}, \mathit{context}_{\mathit{new}}) \end{array}}{\pi, \theta \vdash c \to_{\mathit{C\text{-}IL+MASM}} c\left[\,ac := \mathit{context}_{\mathit{new}},\ sc := ([\![\&e_0]\!]_c^{\theta,\pi}, \mathit{regs}_{\textbf{callee}}, \mathit{inc}_{\mathit{loc}}(s)) \circ c.sc\,\right]} \quad \text{(C-IL-TO-MASM)}$$

$$\frac{\begin{array}{c} c.ac = (\mathit{regs}, s) \qquad ext(c) \qquad \mathit{instr}_{\mathit{next}}(s, \pi.\pi_{\mathit{MASM}}) = \textbf{call}\ p \\ \pi.\pi_{\mathit{C\text{-}IL}}(p).\mathit{npar} - 4 \leq \textbf{hd}(s).\mathit{lifo} \qquad s' = \mathit{drop}_{\mathit{lifo}}(s, \pi(p).\mathit{npar} - 4) \qquad \mathit{MASM2CIL}_{\mathit{ctxt}}(c, p, \textbf{hd}(s).\mathit{lifo}, \mathit{context}_{\mathit{new}}) \end{array}}{\pi, \theta \vdash c \to_{\mathit{C\text{-}IL+MASM}} c\,[\,ac := \mathit{context}_{\mathit{new}},\ sc := (\mathit{regs}|_{\mathcal{R}_{\textbf{callee}}}, \mathit{inc}_{\mathit{loc}}(s')) \circ c.sc\,]}$$
(MASM-TO-C-IL)

$$\frac{\begin{array}{c} c.ac = s \qquad ext(c) \qquad \mathit{stmt}_{\mathit{next}}(s, \pi.\pi_{\mathit{C\text{-}IL}}) = \textbf{return}\ e \\ \textbf{hd}(c.sc) = (\mathit{regs}_{\textbf{callee}}, s') \qquad \mathit{regs}'|_{\mathcal{R}_{\textbf{callee}}} = \mathit{regs}_{\textbf{callee}} \qquad \mathit{regs}'(R_{rv}) = \mathit{val2bytes}([\![e]\!]_c^{\pi,\theta}) \end{array}}{\pi, \theta \vdash c \to_{\mathit{C\text{-}IL+MASM}} c\,[\,ac := (\mathit{regs}', s'),\ \ sc := \textbf{tl}(c.sc)\,]}$$
(RETURN-C-IL-TO-MASM)

$$\frac{\begin{array}{c} c.ac = (\mathit{regs}, s) \qquad ext(c) \qquad \mathit{instr}_{\mathit{next}}(s, \pi.\pi_{\mathit{MASM}}) = \textbf{ret} \\ \textbf{hd}(c.sc) = (\mathit{rds}, \mathit{regs}_{\textbf{callee}}, s') \qquad \mathit{regs}|_{\mathit{dom}(\mathit{regs}_{\textbf{callee}})} = \mathit{regs}_{\textbf{callee}} \qquad c' = \mathit{write}^{\theta}_{\mathit{raw}}((c.\mathcal{M}, s'), \mathit{rds}, \mathit{regs}(R_{rv})) \end{array}}{\pi, \theta \vdash c \to_{\mathit{C\text{-}IL+MASM}} c\,[\,\mathcal{M} := c'.\mathcal{M},\ \ ac := c'.s,\ \ sc := \textbf{tl}(c.sc)\,]}$$
(RETURN-MASM-TO-C-IL)

**Table 8.** Representative choice of transitions from the semantics of *C-IL+MASM*

*Pure Steps* For $\neg ext(x)$, we have a pure step: We simply perform a step of the top-most execution context according to the semantics of the corresponding language.

*Call from C-IL to MASM* For an external call from *C-IL* to *MASM*, a new *MASM* context is created and initialized according to the calling convention. The currently active *C-IL* context is retired to the list of inactive contexts. Constraints on the nondeterministically-chosen new context and the callee-save registers expected by the now inactive *C-IL*-context are captured in the predicate $\mathit{CIL2MASM}_{\mathit{ctxt}}(c, f, E, \mathit{regs}_{\textbf{callee}}, \mathit{context}_{\mathit{new}})$:

$$\begin{aligned} \mathit{context}_{\mathit{new}}.s &= [\mathit{frame}_{\mathit{new}}] \\ \mathit{context}_{\mathit{new}}.\mathit{regs}(R_{p_i}) &= [\![E[i-1]]\!]_c^{\theta,\pi} \quad \text{ if } 1 \leq i \leq 4\ \wedge\ i \leq \mathit{npar} \end{aligned}$$

where *npar* is a shorthand that denotes $\pi.\pi_{\mathit{MASM}}(f).\mathit{npar}$ and $[\![e]\!]_c^{\theta,\pi} \overset{def}{=} [e]^{\theta,\pi.\pi_{\mathit{C\text{-}IL}}}_{(c.\mathcal{M},c.ac)}$. The stack of the new active *MASM*-context consists of a single frame $\mathit{frame}_{\mathit{new}}$:

$$\mathit{frame}_{\mathit{new}}.p = f, \qquad \mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \qquad \mathit{frame}_{\mathit{new}}.\mathit{lifo} = []$$

$$\mathit{frame}_{\mathit{new}}.\mathit{pars}[i] = [\![E[i]]\!]_c^{\theta,\pi}, \qquad 4 \leq i < \mathit{npar}$$

Register content is chosen nondeterministically except for the values of the registers $R_{p_1}, \ldots, R_{p_4}$ (parameters passed in registers according to the calling convention). The remaining parameters are passed on the stack. As a final constraint, the callee-save registers expected by the retired *C-IL*-execution-context are the same as in the new *MASM*-execution-context:

$$\mathit{regs}_{\textbf{callee}} = \mathit{context}_{\mathit{new}}.\mathit{regs}|_{\mathcal{R}_{\textbf{callee}} \setminus \pi.\pi_{\mathit{MASM}}(f).\mathit{uses}}$$

Note: since the *MASM*-compiler guarantees that registers in the uses list will be stored and restored properly, we only have to consider the remaining callee-save registers.

*Return from MASM to C-IL* When returning, callee-save registers must have the values expected by the *C-IL*-context we return to. The content of the return-value register $R_{rv}$ is written to the return destination *rds* given in the *C-IL*-context we return to.

*Call from MASM to C-IL* Calling from *MASM* to *C-IL*, we create a new active *C-IL*-execution-context and transfer the currently active execution context to the list of inactive contexts. $\mathit{MASM2CIL}_{\mathit{ctxt}}(c, p, \mathit{lifo}, \mathit{context}_{\mathit{new}})$ enforces the following constraints:

$$\mathit{context}_{\mathit{new}}.s = [\mathit{frame}_{\mathit{new}}]$$

where

$$\mathit{frame}_{\mathit{new}}.f = p, \quad \mathit{frame}_{\mathit{new}}.\mathit{loc} = 0, \quad \mathit{frame}_{\mathit{new}}.\mathit{rds} = \bot$$

$$\forall \mathit{npar} \leq i < \mathbf{len}(\mathcal{V}) : \quad \mathbf{len}(\mathit{frame}_{\mathit{new}}.\mathcal{M}_{\mathcal{E}}(v_i)) = \mathit{size}(t_i)$$

$$\forall 0 \leq i \leq 3 : (i+1) \leq \mathit{npar} \Rightarrow \mathit{read}^{\theta}_{\mathcal{E}}(\mathit{frame}_{\mathit{new}}, v_i, 0, t_i) = \mathit{bytes2val}(\mathit{regs}(R_{p_{i+1}}), t_i)$$

$$\forall 4 \leq i < \mathit{npar} : \mathit{read}^{\theta}_{\mathcal{E}}(\mathit{frame}_{\mathit{new}}, v_i, 0, t_i) = \mathit{bytes2val}(\mathit{lifo}[\mathbf{len}(\mathit{lifo}) - 1 - i], t_i)$$

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.

*Return from C-IL to MASM* Callee-save registers of the restored *MASM*-context stay the same, the remaining registers get assigned nondeterministically, except for $R_{rv}$. We convert the result of evaluation of the return expression $[\![e]\!]^{\theta,\pi}_{c}$ to its byte-representation and assign this value to the return value register $R_{rv}$.

## 5 Pervasive Theory

To gain a correctness result over the whole system consisting of hard- and software, we apply pervasive verification. In the pervasive stack sketched in Figure 1, adjacent models are always connected by means of simulation theorems in such a way that the abstraction provided by a higher-up layer is sound with respect to the corresponding lower layer. Our reduction theorems involve placing additional assumptions on the lower layer's execution that allow us to construct a more abstract upper layer. We only use assumptions that we can explicitly guarantee for the code we consider.

We are interested in justifying that our combined *C-IL+MASM*-semantics indeed correctly captures the behavior of the underlying *ISA-Assembler*-execution. In order to prove this, we make use of compiler correctness for *C-IL* and *MASM*. In particular, we rely on an explicit simulation relation that connects to the underlying *ISA-Assembler*-model (we use the term *compiler consistency relation* to refer to it).

For pure steps, we can simply apply the assumed compiler correctness. Only for inter-language steps, we need to prove that, given a state in which the compiler simulation holds for the current language, the corresponding compiler consistency relation

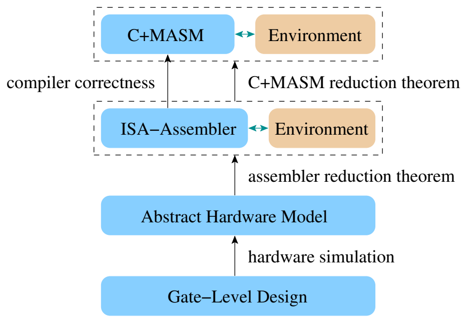

**Fig. 1.** Model stack for operating systems and hypervisor verification.

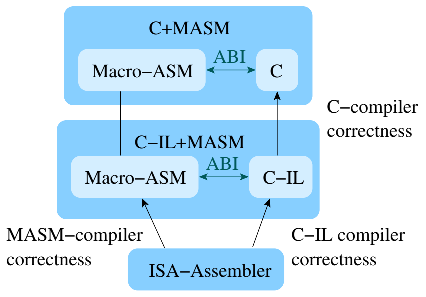

**Fig. 2.** Close-up view of the model stack.

is established. That is, after executing the compiled code of the external call, we reach an *ISA-Assembler*-configuration which is consistent with the the configuration of the abstract machine that has performed the call.

## 5.1 Compiler Consistency

In the actual formal definitions of the relations described in the following, we applied earlier results from the Verisoft project [6].

**Memory & Code Consistency** We consider an *ISA-Assembler*-memory and a *C-IL+MASM*-memory to be consistent iff they carry identical values on all addresses except for those from the stack- or the code-region. The code region contains the compiled code of the program.

***MASM*: Register Consistency** Configurations are consistent iff all registers of the active *MASM*-execution-context except those we abstracted away (instruction pointer, stack pointers) are the same in the *ISA-Assembler*-configuration.

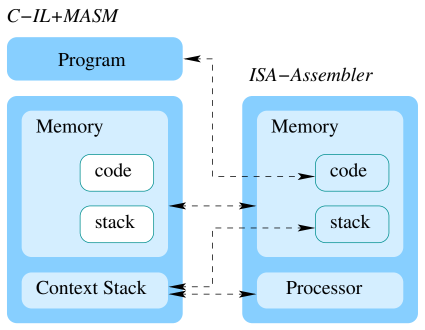

**Fig. 3.** Simulation of *C-IL+MASM* by the underlying *ISA-Assembler*-model.

***MASM*: Stack Consistency** Stack consistency describes on the one hand how the abstract stack matches the stack region in an *ISA-Assembler*-configuration, i.e., for the stacks of the *MASM*-contexts, how *pars*, *saved*, and *lifo* are laid out in memory and pointed to by the stack pointer registers. On the other hand, it also relates the function name and location pairs found in the abstract stack frames to the instruction pointer register, respectively the return address fields in the concrete stack layout.

***C-IL*: Stack Consistency** For the *C-IL*-part, this describes how local memories $\mathcal{M}_{\mathcal{E}}$ from *C-IL*'s stack frames are represented in the concrete stack layout and in the processor registers (parameters in registers, register allocation for local variables). The base address of the return destination *rds* is part of the frame header. As in the *MASM*-case, control consistency is expressed over the function name/location pairs occurring in it.

**Software Conditions**

*No Explicit Writes to Stack and Code Region* Since we explicitly manage a stack abstraction, bypassing this and writing directly to the memory region occupied by the stack will break stack consistency. Also, we do not consider semantics of self-modifying code, so the code region shall never be written. To use the described semantics, this property must be guaranteed (e.g. by performing formal verification).

*No Explicit Update of Stack Pointers* For maintaining a consistent configuration (w.r.t. to the assembler execution of the compiled code) for these semantics, we should never explicitly update the stack pointers. All changes to them currently happen automatically, as a part of the $push$, $pop$, $call$ and $ret$ execution of *MASM*.

## 6 Results

The *C-IL+MASM*-model described here has been applied to extend the baby hypervisor verification results obtained earlier [14]. In those results, there was merely specification of the effect of assembler code for the context switch between guest and hypervisor – with the presented theory, this gap has been closed. Compilation rules from *MASM* to *ISA-Assembler* and compiler consistency relations for both *C-IL* and *MASM* have been specified in full detail for the reduced version of the VAMP-processor used in baby hypervisor verification.

## 7 Future Work

The ideas presented can be applied such that the resulting semantics can serve as a basis for soundness proofs of translation-based assembler verification approaches, as the one described and implemented by Stefan Maus in the Vx86-tool [13].

In order to have a model on which threading libraries can be verified (including the assembler portion that actually performs the stack switch), the presented theory can be be extended to allow stack pointer updates. This work is currently in progress.

For multi-core machines, we need to consider store-buffers. Integrating the results of Cohen and Schirmer on store-buffer reduction [15] appears to be a useful step.

In order to prove correct execution of compiled code in a multi-core context, we need to place some restrictions on memory accesses in order to justify that interleaving instructions on *C-IL+MASM*-level is actually sound with respect to the underlying execution model. This work is currently in progress using an explicit ownership-model.

In an operating system and hypervisor verification effort, interrupts cannot be neglected. There is work in progress to extend the pervasive theory in such a way that interrupt handlers can be seen as additional threads interleaved with regular execution.

## 8 Conclusion

We have presented an integrated semantics of a simple *C*-intermediate-language and a high-level assembler language. Choosing identical memory models and stack abstraction in form of lists of stack frames makes this integration very simple. Distinguishing formally between active and inactive execution contexts, we are able to precisely model the calling conventions between *C-IL* and *MASM*. Based on earlier results, we specified compiler consistency relations for *C-IL* and *MASM* to justify that the integrated semantics presented is a sound abstraction of execution of the compiled code.

## References

1. Klein, G., et al.: seL4: Formal verification of an OS kernel. In: Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, USA, ACM (2009) 207–220
2. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2007) 97–108
3. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2006) 320–333
4. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs. (2007) 189–206
5. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. J. Autom. Reasoning **42**(2-4) (2009) 301–347
6. Leinenbach, D., Petrova, E.: Pervasive compiler verification – from verified programs to verified systems. In: 3rd intl Workshop on Systems Software Verification (SSV08), Elsevier Science B. V. (2008)
7. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning **43**(3) (2009) 263–288
8. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115
9. Appel, A.W.: Verified software toolchain. In: Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software. ESOP'11/ETAPS'11, Springer-Verlag (2011)
10. Morrisett, J.G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. In: Types in Compilation. (1998) 28–52
11. Gurevich, Y., Huggins, J.K.: The semantics of the C programming language. In: Computer Science Logic, volume 702 of LNCS, Springer (1993) 274–308
12. Papaspyrou, N.S.: A formal semantics for the C programming language. tech. report. (1998)
13. Maus, S., Moskał, M., Schulte, W.: Vx86: x86 assembler simulated in C powered by automated theorem proving. In: (AMAST 2008), LNCS 5140. (2008)
14. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: Proceedings of the Third international conference on Verified software: theories, tools, experiments. VSTTE'10, Berlin, Heidelberg, Springer-Verlag (2010) 40–54
15. Cohen, E., Schirmer, N.: A better reduction theorem for store buffers. CoRR **abs/0909.4637** (2009)