Combining Domain-Specific and Foundational Logics to Verify Complete Software Systems

Xinyu Feng[1], Zhong Shao[2], Yu Guo[3], and Yuan Dong[4]

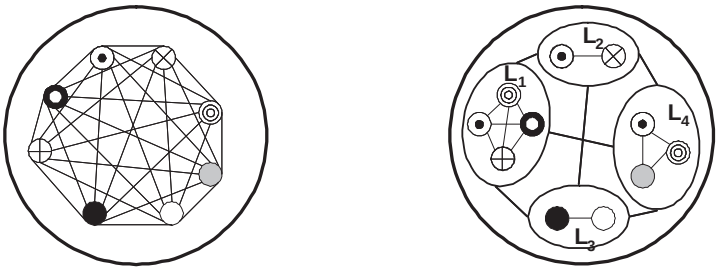
[1]Toyota Technological Institute at Chicago, [2]Yale University
[3]University of Science and Technology of China, [4]Tsinghua University

Abstract. A major challenge for verifying *complete* software systems is their complexity. A complete software system consists of program modules that use many language features and span different abstraction levels (*e.g.,* user code and run-time system code). It is extremely difficult to use one verification system (*e.g.,* type system or Hoare-style program logic) to support all these features and abstraction levels. In our previous work, we have developed a new methodology to solve this problem. We apply specialized "domain-specific" verification systems to verify individual program modules and then link the modules in a foundational open logical framework to compose the verified complete software package. In this paper, we show how this new methodology is applied to verify a software package containing implementations of preemptive threads and a set of synchronization primitives. Our experience shows that domain-specific verification systems can greatly simplify the verification process of low-level software, and new techniques for combining domain-specific and foundational logics are critical for the successful verification of complete software systems.

1 Introduction

It is difficult to verify complete software systems because they use many different language features and span different abstraction levels. As an example, our ongoing project to verify a simplified operating system kernel exposes such challenges. The kernel has a simple bootloader, kernel-level threads and a thread scheduler, synchronization primitives, hardware interrupt handlers, and a simplified keyboard driver. Although it has only around 1,300 lines of x86 assembly code, it already uses features such as dynamic code loading, thread scheduling and context switching, concurrency, hardware interrupts, device drivers and I/O. *How do we verify safety and correctness of the whole system with machine-checkable proofs?*

Verifying the whole system in a single type system or program logic is impractical because, as Fig. 1 (a) shows, such a verification system needs to consider all possible interactions among these different features, many of which are even at different abstraction levels. The resulting logic, if exists, would be very complex and difficult to use. Fortunately, in reality, programmers seem to never use all the features at the same time. Instead, only limited combinations of features—at certain abstraction level—are used in individual program modules. It would be much simpler to design and use specialized "domain specific" logics to verify individual program modules, as shown in Fig. 1 (b). To allow interactions of modules and build a complete system, we need to also support the interoperability of different logics.



(a). The One-for-All Logic (b). Domain-Specific Logics

Fig. 1. Using Domain Specific Logics to Verify Modules

In our previous work [6], we proposed a new methodology for modular verification and linking of low-level software systems. Different type systems and program logics can be used to verify different program modules. These verification systems (called *domain-specific systems* or *foreign systems*) are then embedded into a foundational open framework, OCAP, where interoperability is supported. Verified program modules in a foreign system can be reused in the framework, given a sound embedding of the foreign system. Safety properties of the whole program can be derived from verified modules.

In this paper, we show how to use this methodology to verify the implementations of preemptive threads and a set of synchronization primitives extracted from our OS kernel. As shown in Fig. 2, the modules include thread scheduling, blocking and unblocking, a timer interrupt handler, implementations of locks and condition variables, and yield and sleep primitives. To verify them, we develop OCAP-x86, which adapts the OCAP framework for the x86 architecture and extends it with the support of hardware interrupts. We then introduce four domain-specific program logics, each designed for a specific set of program modules that share the same abstraction level. These logics are all embedded into OCAP-x86. The soundness of the embedding and the interoperability of foreign logics are formally proved in the Coq proof assistant [4]. Program modules are verified in "domain-specific" foreign logics with machine-checkable proofs.

This verification effort depends critically on our previous work on the OCAP theories [6] and specialized program logics for reasoning about interrupts [7] and runtime stacks [9]. In this paper, instead of developing new theories or logics, we present a case study of our methodology and report our experience in applying it to fully verify a relatively complex software package. Specifically, our experience shows that our verification project can benefit from this new methodology in many different ways.

1. Implementations of thread primitives are at a lower abstraction level than synchronization primitives. It is very natural to verify them using different logics.
2. Using specialized logics, we can hide irrelevant details about the environment from the specification of program modules, and greatly simplify the inference rules for reasoning about programs. These details are added back when the foreign logics are embedded into OCAP-x86. This improves the modularity and the reusability of the verified primitives.
3. By abstracting away the thread implementation details in the logics used to verify synchronization primitives, we avoid supporting first-class code pointers in these

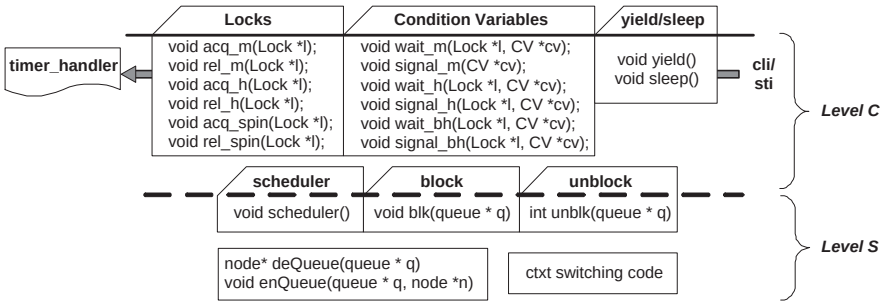


Fig. 2. The Thread Implementations

logics, even though the underlying thread primitives treat program counters saved in thread control blocks (TCBs) as normal data.

4. Using specialized logics is an effective way to ban some instructions in certain specific modules. For instance, the "iret" instruction is only used to "return" from interrupt handlers to non-handler code, and should not be used outside handlers.

In the rest of this paper, we first show in Sect. 2 the challenges to verify the package and give an overview of our solutions. We present our modeling of the x86 assembly language in Sect. 3 and the OCAP-x86 framework in Sect. 4. We then explain in detail the verification of thread implementations in Sect. 5 and synchronization primitives in Sect. 6. We give more details of our Coq implementations in Sect. 7, and discuss about related work and conclude in Sect. 8.

2 Background and Challenges

The program modules in Fig. 2 are extracted from our simplified OS kernel (for uniprocessor platforms only) which is implemented in 16-bit x86 assembly code and works in real mode. The scheduler saves the execution context of the current thread into the ready queue, picks another one from the queue, and switches to the execution context of the new thread. block takes a pointer to a block queue as argument, puts the current thread into the block queue, and switches the control to a thread in the ready queue. unblock also takes a pointer to a block queue as argument; it moves a thread from the block queue to the ready queue but does not do context switching. Execution of threads may be interrupted by hardware interrupts. When the interrupt request comes, the control is transferred to the interrupt handler. To implement preemptive threads, the handler calls scheduler before it returns. Threads can also yield the control voluntarily by calling scheduler. block and unblock are used to implement synchronization primitives.

There are many challenges to fully verify these program modules. Our previous work has solved problems related to specific language features, such as control abstractions [9] and preemptive threads with hardware interrupts [7]. However, to verify all the modules as part of a whole system, we still need to address the following problems.

How to verify thread implementations? The code implementing synchronization primitives is concurrent because it may be run by different threads and be preempted by interrupt handlers. It needs to be verified using a concurrent program logic. However, it

is difficult to use the same logic to verify scheduler, block and unblock. Concurrent program logics, *e.g.,* rely-guarantee reasoning [11] or Concurrent Separation Logic [15], usually assume built-in concurrency and abstract away schedulers. The scheduler and the block primitives, on the other hand, switch the execution contexts of threads. They are at a lower abstraction level than threads.

Another problem is that the thread primitives manipulate threads' execution contexts containing program counters pointing to the code where threads resume execution. These code pointers are stored in memory, thus can be read and updated as normal data. To verify thread implementations using a concurrent program logic, we need to support "first-class" code pointers, which is difficult in Hoare-style reasoning and the solutions are usually heavyweight [13].

How to enforce local invariants? Program invariants play an important role in verification. Knowing that the invariants always hold, we can use them without putting them in specifications, which simplifies verification steps and improves modularity. Although there is no meaningful system-wide invariants other than the most basic safety requirements, more refined invariants are preserved locally in different program modules. How do we take advantage of these local invariants and ensure that they are preserved in corresponding modules? For instance, to avoid race conditions, the implementation of scheduler, block and unblock needs to disable interrupts. Therefore the potential interrupts by interrupt handlers should never be a concern when verifying these subroutines. Another example is that thread code, including implementations of synchronization primitives, assumes thread queues are well-formed. Also the well-formedness is trivially preserved because the thread code never directly accesses these queues other than through thread primitives. Since these invariants are only preserved locally in individual modules, it is difficult to enforce them globally in the program logic.

How to ban certain instructions in specific modules? As mentioned before, interrupts are not concerned in thread primitives, assuming they are always disabled. However, this invariant cannot be preserved if the sti instruction is executed, which enables interrupts. Actually thread implementations should only use a subset of instructions and are not supposed to execute any interrupt-related ones. Similarly, the iret instruction in x86 is the last instruction of interrupt handlers. It returns control to the non-handler code. This instruction cannot be used interchangeably with the normal ret instruction for functions, and should not be used outside of interrupt handlers. We need to have a way to ban these instructions in specific scenarios.

To solve these problems, we verify the package following the open-framework-based methodology. As shown in Fig. 2, we split the code into two layers: the upper level C (for "Concurrent") and the low level S (for "Sequential"). Code at Level C is concurrent; it handles interrupts explicitly and implements interrupt handlers but abstracts away the implementation of threads. Code at Level S is sequential (always executed with interrupt disabled); functions that need to know the concrete representations of thread control blocks (TCBs) and thread queues are implemented at Level S.

We use a Hoare-style program logic for sequential assembly code to verify Level S. Knowing that interrupts are always disabled, specifications for Level S do not need to mention the fact at all. To ban interrupt-related instructions at this level, we simply exclude any rules for these instructions in the specialized logic, so code containing

$$
\begin{array}{rrcl}
\textit{(World)} & \mathbb{W} & ::= & (\mathbb{C}, \mathbb{S}, \mathtt{pc}) \\
\textit{(Code)} & \mathbb{C} & ::= & \{\mathtt{f}_1 \rightsquigarrow \mathtt{c}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{c}_n\} \\
\textit{(State)} & \mathbb{S} & ::= & (\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}) \\
\textit{(Mem)} & \mathbb{M} & ::= & \{\mathtt{l}_1 \rightsquigarrow \mathtt{w}_1, \ldots, \mathtt{l}_n \rightsquigarrow \mathtt{w}_n\} \\
\textit{(RegFile)} & \mathbb{R} & ::= & \{\mathsf{ax} \rightsquigarrow \mathtt{w}_1, \ldots, \mathsf{sp} \rightsquigarrow \mathtt{w}_8\} \\
\textit{(FlagReg)} & \mathbb{F} & ::= & \{\mathsf{if} \rightsquigarrow \mathtt{b}_1, \mathsf{zf} \rightsquigarrow \mathtt{b}_2\}
\end{array}
$$

$$
\begin{array}{rrcl}
\textit{(Word)} & \mathtt{w} & ::= & i \ (\textit{nat numbers}) \\
\textit{(Labels)} & \mathtt{l}, \mathtt{f}, \mathtt{pc} & ::= & i \ (\textit{nat numbers}) \\
\textit{(Boolean)} & \mathtt{b}, \mathsf{isr} & ::= & \mathtt{tt} \mid \mathtt{ff} \\
\textit{(Addr)} & \mathtt{d} & ::= & i \mid i(\mathtt{r}) \\
\textit{(Oprand)} & \mathtt{o} & ::= & i \mid \mathtt{r}
\end{array}
$$

$$
\begin{array}{rrcl}
\textit{(Reg)} & \mathtt{r} & ::= & \mathsf{ax} \mid \mathsf{bx} \mid \mathsf{cx} \mid \mathsf{dx} \mid \mathsf{bp} \mid \mathsf{di} \mid \mathsf{si} \mid \mathsf{sp} \\
\textit{(Instr)} & \mathtt{\iota} & ::= & \mathsf{add}\ \mathtt{r}, \mathtt{o} \mid \mathsf{sub}\ \mathtt{r}, \mathtt{o} \mid \mathsf{mov}\ \mathtt{r}, \mathtt{o} \mid \mathsf{mov}\ \mathtt{r}, \mathtt{d} \mid \mathsf{mov}\ \mathtt{d}, \mathtt{r} \mid \mathsf{cmp}\ \mathtt{r}, \mathtt{o} \mid \mathsf{jne}\ \mathtt{f} \\
 & & & \mid \mathsf{push}\ \mathtt{o} \mid \mathsf{push}\ \mathtt{d} \mid \mathsf{pushf} \mid \mathsf{pop}\ \mathtt{o} \mid \mathsf{pop}\ \mathtt{d} \mid \mathsf{popf} \mid \mathsf{sti} \mid \mathsf{cli} \mid \mathsf{eoi} \mid \mathsf{call}\ \mathtt{f} \\
\textit{(Command)} & \mathtt{c} & ::= & \mathtt{\iota} \mid \mathsf{jmp}\ \mathtt{f} \mid \mathsf{ret} \mid \mathsf{iret}
\end{array}
$$

Fig. 3. The Machine

these instructions cannot be verified. It is also interesting to note that we do not need to support first-class code pointers in this level to verify context switching code, because program counters saved in TCBs are treated as normal data by thread primitives. Although they point to code at Level C, Level C is not verified *within* this logic.

We hide the implementation of threads from Level C. To do this, we design an abstract machine for the higher level code, which treats scheduler, block, and unblock as abstract primitive instructions. Thread queues are abstract algebraic structures (*e.g.,* sets) instead of data structures in memory. They are inaccessible by normal instructions except scheduler, block and unblock. Various program logics are used to verify modules at this level. In these logics, we do not need to specify the well-formedness of thread queues in memory. This not only improves the modularity and reusability of verified modules, but avoids the support of first-class code pointers because code pointers in the thread queues are no longer first-class in this abstract machine.

3 The Machine

Figure 3 shows our modeling of a subset of 16-bit x86 machines. The machine configuration contains the code $\mathbb{C}$, the program state $\mathbb{S}$ and the program counter pc. The code maps code labels to commands. The program state contains the memory, the register file, the flag register and a special register isr.

There are several simplifications in our modeling of the x86 architecture. The code $\mathbb{C}$ is not part of the heap and is read-only. The von Neumann architecture can be supported following Cai *et al.* [3]. We do not model the full PIC (programmable interrupt controller). Instead, we assume only the timer interrupt is allowed and all other interrupts are masked. The isr register, which is now degenerated into a binary value, records whether the timer interrupt is currently being handled. If it is set, new interrupt requests will not be processed. The bit needs to be cleared before the interrupt handler returns. The instruction eoi clears isr. It is a shorthand for the real x86 instruction "out dx, al", with the precondition that ax and dx both contain value 0x20.

The if flag records whether interrupts are enabled. It can be set by sti and cleared by cli. At each program point, the control is transferred to the entry point of the interrupt handler if there is an incoming request and the interrupt is enabled. Otherwise the

$$
\begin{array}{rrcl}
(\mathit{CHSpec}) & \Psi & \in & \mathcal{P}(\mathit{Labels} * \mathit{OCdSpec}) \\
(\mathit{OCdSpec}) & \pi & ::= & \langle \rho, \mathcal{L}, \theta \rangle \quad \in \mathit{LangID} * (\Sigma X : \mathsf{Type}.\, X) \\
(\mathit{LangID}) & \rho & ::= & n\ (\mathit{nat\ nums}) \\
(\mathit{LangTy}) & \mathcal{L} & ::= & (\mathit{CiC\ terms}) \quad \in \mathsf{Type} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathit{CiC\ terms}) \quad \in \mathcal{L} \\
(\mathit{Assert}) & \mathtt{a} & \in & \mathit{CHSpec} \to \mathit{State} \to \mathsf{Prop} \\
(\mathit{Interp}) & [\![\_]\!]_{\mathcal{L}} & \in & \mathcal{L} \to \mathit{Assert} \\
(\mathit{LangDict}) & \mathcal{D} & \in & \mathit{LangID} \rightharpoonup \Sigma X : \mathsf{Type}.\, (X \to \mathit{Assert})
\end{array}
$$

Fig. 4. Specification Constructs of OCAP-x86

next instruction at pc is executed. Instead of modeling the source of the interrupt, we use a non-deterministic semantics and assume the interrupt may come at any moment. Formula (1) formalizes the condition under which an interrupt request will be handled:

$$\mathsf{enable\_itr}(\mathbb{S}) \stackrel{\mathbf{def}}{=} (\mathbb{S}.\mathbb{F}(\mathsf{if}) = \mathtt{tt}) \wedge (\mathbb{S}.\mathsf{isr} = \mathtt{ff})\,. \tag{1}$$

Because we support only the timer interrupt, we assume there is a global entry point ih_entry of the interrupt handler. When an interrupt request is handled, the encoding of $\mathbb{F}$ and pc is pushed onto the stack[1]; if is cleared and isr is set; and pc is set to ih_entry. Semantics of other instructions are standard. The formal definition of operational semantics can be find in our Coq implementation [8] (the file `Operational_Semantics.v`).

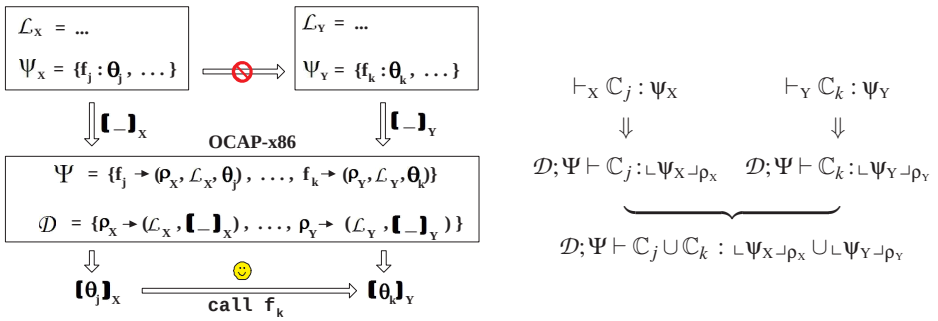
4 The OCAP-x86 Framework

OCAP-x86 adapts OCAP [6] for our x86 machine and extends it to support interrupts. It is developed in Coq [4], which implements CiC [17] and supports mechanized theorem proving in Higher-Order Logic. In Coq, the universe of logical propositions is Prop. Coq can also be used as a meta-language so that domain-specific logics can be defined using inductive definitions. These inductively defined types in Coq are in universe Type.

In OCAP-x86, the code heap specification Ψ for $\mathbb{C}$ associates code labels with generic specifications π, which encapsulate concrete specifications in domain-specific logics into a uniform format. Each π is a triple containing an identifier ρ of a domain-specific logic, the CiC meta-type $\mathcal{L}$ of its specification language, and a concrete specification θ in $\mathcal{L}$. In OCAP-x86, specifications from domain-specific logics are mapped to a foundational form a, which is a logical predicate over Ψ and program states $\mathbb{S}$. For each $\mathcal{L}$, the mapping is done by the interpretation $[\![\_]\!]_{\mathcal{L}}$. The dictionary $\mathcal{D}$ of specification languages then maps language identifiers ρ to the specification language's meta-type and the corresponding interpretation. We allow one specification language $\mathcal{L}$ to have multiple interpretations, each with a unique ρ in $\mathcal{D}$.

$$\frac{\begin{array}{l}(\mathsf{ih\_entry}, \pi_0) \in \Psi \quad \forall \Psi', \mathbb{S}.\ \mathtt{a}\ \Psi'\ \mathbb{S} \wedge \Psi \subseteq \Psi' \\ \qquad \to \exists \mathtt{f}', \mathbb{S}', \pi'.\ (\mathtt{f}' = \mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})) \wedge (\mathbb{S}' = \mathsf{next}_{(\mathtt{f},\iota)}(\mathbb{S})) \\ \qquad\qquad \wedge (\mathtt{f}', \pi') \in \Psi \wedge [\![\pi']\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}' \\ \qquad\qquad \wedge (\mathsf{enable\_itr}(\mathbb{S}) \to \exists \mathbb{S}' = \mathsf{next\_itr}(\mathbb{S}, \mathtt{f}).\ [\![\pi_0]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}')\end{array}}{\mathcal{D}; \Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \iota} \ (\textsc{ins})$$

[1] Some arbitrary value is also pushed onto stack as the "cs" register.



(a) Embedding (b) Linking

Fig. 5. Interfacing and Linking Modules

The INS rule just shown is a schema in OCAP-x86 for all instructions. Informally, the judgment $\mathcal{D};\Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \iota$ says if the precondition $\mathtt{a}$ holds, it is safe to execute the instruction at the label $\mathtt{f}$ and the resulting state satisfies the post-condition in Ψ (which is also the precondition of the following computation). $\mathcal{D}$ is used to interpret specifications in Ψ. $\mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})$ and $\mathsf{next}_{(\mathtt{f},\iota)}(\mathbb{S})$ define the new pc and state after executing ι at $\mathtt{f}$ with the initial state $\mathbb{S}$. The specification at $\mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})$ in Ψ is used as the post-condition. In addition to the sequential execution, we also need to consider the interrupt if $\mathsf{enable\_itr}(\mathbb{S})$ holds. In this case, the next state needs to satisfy the precondition π_0 for the interrupt handler. $\mathsf{next\_itr}(\mathbb{S})$ is the next state after the interrupt comes at state $\mathbb{S}$. $\mathsf{next\_pc}_{(\iota,\mathbb{S})}(\mathtt{f})$, $\mathsf{next}_{(\mathtt{f},\iota)}(\mathbb{S})$ and $\mathsf{next\_itr}(\mathbb{S})$ are part of the operational semantics and the details are omitted here. $[\![\pi]\!]_{\mathcal{D}}$ is the interpretation of π in $\mathcal{D}$, which is defined as:

$$[\![\langle \rho, \mathcal{L}, \theta\rangle]\!]_{\mathcal{D}} \stackrel{\text{def}}{=} \lambda\Psi,\mathbb{S}.\ \exists [\![\_]\!]_{\mathcal{L}}.\ \mathcal{D}(\rho) = (\mathcal{L}, [\![\_]\!]_{\mathcal{L}}) \wedge [\![\theta]\!]_{\mathcal{L}}\ \Psi\ \mathbb{S}. \tag{2}$$

The LINK rule below links separately verified modules $\mathbb{C}_1$ and $\mathbb{C}_2$.

$$\frac{\mathcal{D}_1;\Psi_1 \vdash \mathbb{C}_1 : \Psi_1' \quad \mathcal{D}_2;\Psi_2 \vdash \mathbb{C}_2 : \Psi_2' \quad \mathcal{D}_1 \# \mathcal{D}_2 \quad \mathbb{C}_1 \# \mathbb{C}_2}{\mathcal{D}_1 \cup \mathcal{D}_2;\Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1' \cup \Psi_2'}\ (\text{LINK})$$

We use $f \# f'$ to mean the partial functions f and f' maps the same value to the same image. The judgment $\mathcal{D};\Psi \vdash \mathbb{C} : \Psi'$ means the module $\mathbb{C}$ is verified with imported interface Ψ and exported interface Ψ'. The complete set of OCAP-x86 rules is presented in the companion technical report [8].

How to use OCAP-x86. Inference rules of OCAP-x86 are designed for generality with minimum constraints instead of for ease of use. Actually the rules are not intended to be used directly by program verifiers. Instead, domain-specific logics should be designed above the foundational framework to verify specific modules. The specifications and rules in the domain-specific logics can be customized and simplified, given the specific local invariants preserved in the modules.

Figure 5(a) shows two domain-specific logics with different specification languages $\mathcal{L}_{\mathrm{X}}$ and $\mathcal{L}_{\mathrm{Y}}$. Each has its own customized rules. To prove the customization is sound, we encode the local invariants in the interpretations $[\![\_]\!]_{\mathrm{X}}$ and $[\![\_]\!]_{\mathrm{Y}}$, and show premises of

the customized rules imply premises of the corresponding OCAP-x86 rules, thus these domain-specific rules are admissible in the foundational framework.

Suppose we have modules verified in $\mathcal{L}_X$ and $\mathcal{L}_Y$, with specifications ψ_X and ψ_Y respectively. They cannot be interfaced directly because ψ_X and ψ_Y are specified in different languages. We map the concrete specifications (*e.g.,* θ_j and θ_k) into the generic triples π in the Ψ at the OCAP-x86 level, where ρ_X and ρ_Y are language IDs assigned to $\mathcal{L}_X$ and $\mathcal{L}_Y$. Their interpretations are put in $\mathcal{D}$. To interface modules, we match in OCAP-x86 the interpreted specifications, which are now assertions $\mathtt{a}$. Since the interpretation usually encodes program invariants, matching the interfaces at this level guarantees the local invariants are compatible at boundaries of modules.

Given the soundness of the embeddings, the modules verified in $\mathcal{L}_X$ and $\mathcal{L}_Y$ can be reused in OCAP-x86 without redoing the proof (see Fig. 5(b), where $\llcorner\psi\lrcorner_\rho$ maps ψ to Ψ in OCAP-x86). Then they can be linked using the LINK rule. The soundness of OCAP-x86 guarantees that the complete system after linking never gets stuck. Also the interpretation of specifications in Ψ holds when the corresponding program points are reached by jmp or call instructions.

Theorem 1 (Soundness of OCAP-x86).
If $\mathcal{D};\Psi \vdash (\mathbb{C},\mathbb{S},\mathtt{pc})$, then, for all n, there exist $\mathbb{S}'$ and $\mathtt{pc}'$ such that $(\mathbb{C},\mathbb{S},\mathtt{pc}) \longmapsto^n (\mathbb{C},\mathbb{S}',\mathtt{pc}')$, and there exists $\Psi' \supseteq \Psi$ such that the following are true:

1. if $\mathbb{C}(\mathtt{pc}') = \mathsf{jmp}\ \mathtt{f}$, there exists π such that $(\mathtt{f},\pi) \in \Psi$ and $[\![\pi]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}'$;
2. if $\mathbb{C}(\mathtt{pc}') = \mathsf{call}\ \mathtt{f}$, there exist π and $\mathbb{S}''$ such that $(\mathtt{f},\pi) \in \Psi, \mathbb{S}'' = \mathsf{next}_{(\mathtt{pc}',\mathsf{call}\ \mathtt{f})}(\mathbb{S}')$, and $[\![\pi]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}''$;
3. if $\mathsf{enable\_itr}(\mathbb{S}')$, there exist π_0 and $\mathbb{S}''$ such that $(\mathsf{ih\_entry},\pi_0) \in \Psi, \mathbb{S}'' = \mathsf{next\_itr}(\mathbb{S}',\mathtt{pc}')$, and $[\![\pi_0]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S}''$.

Here $\mathcal{D};\Psi \vdash \mathbb{W}$ means the whole system $\mathbb{W}$ is well-formed with respect to Ψ (see the TR [8] for its definition); $\mathbb{W} \longmapsto^n \mathbb{W}'$ means an n-step transition from $\mathbb{W}$ to $\mathbb{W}'$.

5 Verifying Thread Implementations

The implementation of threads contains the three thread primitives at Level S in Fig. 2. The scheduler uses the simple FIFO scheduling policy. The thread queues are implemented as doubly-linked lists containing thread control blocks (TCBs). Each TCB contains the status of the thread (Ready, Idle or Blocked) and the value of the sp register when the thread is preempted or blocked. Note that we do not need to store pc in TCB because, by the calling convention, pc is at the top of the stack pointed by sp.

As explained in Sect. 1, the TCB of the running thread is put into the ready queue if it is preempted or it calls scheduler voluntarily. The scheduler sets its status to Ready unless it is an idle thread. When block is called, the current thread is put into the corresponding block queue with the status Blocked. An idle thread with the status Idle would never be blocked. This guarantees the ready queue is not empty when block is called.

To avoid race conditions, code at this level needs to disable interrupts. As a result, the full x86 machine can be simplified into a sequential subset shown in Fig. 6, where anything related to interrupts are removed. It is the machine model in the programmers' mental picture while they are writing and verifying code at Level S. Correspondingly,

$$
\begin{array}{rrcl}
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{M},\mathbb{R},\mathbb{F},\cancel{\mathsf{isr}}) \qquad\qquad (\textit{FlagReg})\ \mathbb{F} ::= \{\cancel{\mathsf{if}} \rightsquigarrow \cancel{\mathtt{b}_1}, \mathsf{zf} \rightsquigarrow \mathtt{b}_2\} \\
(\textit{Instr}) & \mathtt{\iota} & ::= & \mathsf{add}\ \mathtt{r},\mathtt{o} \mid \mathsf{sub}\ \mathtt{r},\mathtt{o} \mid \mathsf{mov}\ \mathtt{r},\mathtt{o} \mid \mathsf{mov}\ \mathtt{r},\mathtt{d} \mid \mathsf{mov}\ \mathtt{d},\mathtt{r} \mid \mathsf{cmp}\ \mathtt{r},\mathtt{o} \mid \mathsf{jne}\ \mathtt{f} \\
 & & & \mid \mathsf{push}\ \mathtt{o} \mid \mathsf{push}\ \mathtt{d} \mid \cancel{\mathsf{pushf}} \mid \mathsf{pop}\ \mathtt{o} \mid \mathsf{pop}\ \mathtt{d} \mid \cancel{\mathsf{popf}} \mid \cancel{\mathsf{sti}} \mid \cancel{\mathsf{cli}} \mid \cancel{\mathsf{eoi}} \mid \mathsf{call}\ \mathtt{f} \\
(\textit{Command}) & \mathtt{c} & ::= & \mathtt{\iota} \mid \mathsf{jmp}\ \mathtt{f} \mid \mathsf{ret} \mid \cancel{\mathsf{iret}}
\end{array}
$$

Fig. 6. A Sequential Subset of x86 Machine

$$
\begin{array}{rrcl@{\qquad}rrcl}
(\textit{StPred}) & \mathtt{p} & \in & \textit{State} \to \mathsf{Prop} & (\textit{LangTy}) & \mathcal{L}_{\mathrm{SCAP}} & ::= & \textit{StPred} * \textit{Guarantee} \\
(\textit{Guarantee}) & \mathtt{g} & \in & \textit{State} \to \textit{State} \to \mathsf{Prop} & (\textit{CdSpec}) & \theta & ::= & (\mathtt{p},\mathtt{g}) \\
 & & & & (\textit{LocalSpec}) & \psi & ::= & \{\mathtt{f}_1 \rightsquigarrow \theta_1, \ldots, \mathtt{f}_n \rightsquigarrow \theta_n\}
\end{array}
$$

Fig. 7. SCAP-x86 Specifications

we use a sequential program logic, SCAP-x86, to verify the thread implementations. SCAP-x86 adapts the SCAP logic [9] to the x86 architectures. The specification constructs are shown in Fig. 7. The specification θ is instantiated using a pair $(\mathtt{p},\mathtt{g})$. $\mathtt{p}$ is a logical predicate in Coq specifying the precondition over the program state; $\mathtt{g}$ is a predicate over a pair of states, which specifies the state transition from the current program point to the end of the current function. The concrete specification ψ of code heaps in SCAP-x86 maps code labels to θ. Note that we do not really implement SCAP-x86 on the sequential machine in Fig. 6. The states specified in $\mathtt{p}$ and $\mathtt{g}$ are still x86 program states defined in Fig. 3. However, $\mathtt{p}$ and $\mathtt{g}$ can leave if and isr unspecified, as if they are specifying the simpler states in Fig. 6. To exclude these interrupts-related instructions, SCAP-x86 simply does not have inference rules for them, therefore any modules using these instructions cannot be verified in SCAP-x86.

Specifications of thread primitives. Figure 8 shows our definitions of abstract thread queues and TCBs. The set of all threads is a partial mapping T from thread IDs to abstract TCBs. Each tcb contains the thread status and the values of sp and $\mathtt{pc}$. Note that, in physical memory, $\mathtt{pc}$ is stored on top of stack instead of in the TCB. We put it in the abstract tcb for convenience of specifications. The abstract queue Q is just a set of thread IDs. B is a partial mapping from block queue IDs to queues for blocked threads.

The specification of $\mathsf{scheduler}$ is $(\mathsf{sch\_pre},\mathsf{sch\_grt})$, as shown in Fig. 9. $\mathsf{sch\_pre}$ is the precondition. It says that T maps the ID of the current thread to its TCB, which has a concrete representation in memory ($\mathsf{embed\_curr\_TCB}(\mathtt{t},\mathsf{tcb})$); the ready queue Q is embedded in memory, storing TCBs in T ($\mathsf{embed\_RdyQ}(\mathsf{T},\mathsf{Q})$); a $\mathtt{pc}$ is at the top of stack pointed by sp ($\mathbb{R}(\mathsf{sp}) = \mathtt{l}$ and $\mathtt{l} \overset{2}{\mapsto} \mathtt{pc}$); and there is sufficient stack space for the scheduler function ($\mathsf{stkSpace}(\mathtt{l},\mathsf{sch\_stk})$, where $\mathsf{sch\_stk}$ is the size of the stack space for $\mathsf{scheduler}$). Here we use separating conjunction in Separation Logic [18] to mean the current TCB, the ready queue and the stack are in different portions of memory. We use $\mathtt{l} \overset{2}{\mapsto} \mathtt{w}$ to mean $\mathtt{w}$ is stored at $\mathtt{l}$ and $\mathtt{l}+1$ (*i.e.,* a 16-bit integer). Also the specification is polymorphic over the part of memory untouched by $\mathsf{scheduler}$, represented by the memory predicate $\mathtt{m}$. We omit the concrete definitions of the embedding of TCBs and thread queues here. Interested readers can refer to our Coq implementation for details [8].

$$
\begin{array}{lll}
(\textit{ThrdSet}) & \mathsf{T} & ::= \{\mathtt{t}_1 \rightsquigarrow \mathsf{tcb}_1, \ldots, \mathtt{t}_n \rightsquigarrow \mathsf{tcb}_n\} \\
(\textit{BlkQSet}) & \mathsf{B} & ::= \{b_1 \rightsquigarrow \mathsf{Q}_1, \ldots, b_n \rightsquigarrow \mathsf{Q}_n\} \\
(\textit{ThrdQ}) & \mathsf{Q} & ::= \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}
\end{array}
\qquad
\begin{array}{lll}
(\textit{Status}) & \mathtt{s} & ::= \mathsf{Ready} \mid \mathsf{Idle} \mid \mathsf{Blocked} \\
(\textit{TCB}) & \mathsf{tcb} & ::= (\mathtt{s}, \mathtt{w}, \mathtt{pc}) \\
(\textit{ThrdID}) & \mathtt{t} & ::= n \text{ (nat nums and } n > 0) \\
(\textit{BQID}) & b & ::= n \text{ (nat nums and } n > 0)
\end{array}
$$

Fig. 8. Abstract Queues and TCBs

$$
\begin{array}{l}
\mathsf{sch\_pre\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathtt{m}) \stackrel{\textbf{def}}{=} \\
\quad \lambda(\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}).\ (\mathsf{T}(\mathtt{t}) = \mathsf{tcb}) \wedge (\mathbb{R}(\mathsf{sp}) = \mathtt{l}) \\
\qquad \wedge (\mathsf{embed\_curr\_TCB}(\mathtt{t}, \mathsf{tcb}) * \mathsf{embed\_RdyQ}(\mathsf{T}, \mathsf{Q}) * (\mathtt{l} \stackrel{2}{\mapsto} \mathtt{pc}) * \mathsf{stkSpace}(\mathtt{l}, \mathsf{sch\_stk}) * \mathtt{m})\ \mathbb{M} \\
\mathsf{sch\_post\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{l}_0, \mathtt{pc}_0, \mathtt{m}) \stackrel{\textbf{def}}{=} \\
\quad \lambda(\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}).\ (\mathsf{T}(\mathtt{t}) = \mathsf{tcb}) \wedge (\mathsf{tcb} = (\_, \mathtt{l}, \_)) \wedge (\mathbb{R}(\mathsf{sp}) = \mathtt{l}) \\
\qquad \wedge (\mathsf{embed\_curr\_TCB}(\mathtt{t}, \mathsf{tcb}) * \mathsf{embed\_RdyQ}(\mathsf{T}, \mathsf{Q}) * (\mathtt{l}_0 \stackrel{2}{\mapsto} \mathtt{pc}_0) * \mathsf{stkSpace}(\mathtt{l}_0, \mathsf{sch\_stk}) * \mathtt{m})\ \mathbb{M} \\
\mathsf{sch\_pre} \stackrel{\textbf{def}}{=} \lambda\mathbb{S}.\ \exists \mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}.\ \mathsf{sch\_pre\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathsf{true})\ \mathbb{S} \\
\mathsf{sch\_grt} \stackrel{\textbf{def}}{=} \lambda\mathbb{S}, \mathbb{S}'.\ \forall \mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathtt{m}.\ \mathsf{sch\_pre\_aux}(\mathsf{T}, \mathtt{t}, \mathsf{tcb}, \mathsf{Q}, \mathtt{l}, \mathtt{pc}, \mathtt{m})\ \mathbb{S} \\
\qquad \rightarrow \exists \mathsf{T}', \mathtt{t}', \mathsf{tcb}', \mathsf{Q}', \mathtt{l}'.\ (\mathsf{T}' = \mathsf{T}\{\mathtt{t} \rightsquigarrow (\mathsf{tcb}.\mathtt{s}, \mathtt{l}, \mathtt{pc})\}) \wedge (\mathsf{Q} \cup \{\mathtt{t}\} = \mathsf{Q}' \cup \{\mathtt{t}'\}) \\
\qquad\qquad \wedge \mathsf{sch\_post\_aux}(\mathsf{T}', \mathtt{t}', \mathsf{tcb}', \mathsf{Q}', \mathtt{l}', \mathtt{l}, \mathtt{pc}, \mathtt{m})\ \mathbb{S}'
\end{array}
$$

Fig. 9. The Specification of scheduler

The auxiliary definition sch_post_aux specifies the post-condition. Here the parameters (T, t, tcb *etc.*) refer to the state immediately before scheduler returns, except $\mathtt{l}_0$ and $\mathtt{pc}_0$, which refer to the value of sp and pc at the beginning of the function. The guarantee sch_grt requires that the pre- and post-conditions hold over states at the beginning and at the end, respectively. It also relates the auxiliary variables in sch_post_aux with those in sch_pre_aux, and ensures that the calling thread of scheduler is added into the ready queue Q; its sp and pc is saved in TCB and is put in T; other TCBs in T are preserved; and the memory specified by m is unchanged. Specifications for block and unblock are in similar forms and not shown here.

SCAP-x86 and the embedding in OCAP-x86. Inference rules in SCAP-x86 are similar to those in the original SCAP for a MIPS-like architecture [9]. Unlike rules in OCAP-x86, where interrupts have to be considered at every step of verification, the SCAP-x86 rules only consider the sequential execution of instructions.

To embed SCAP-x86 into OCAP-x86, we first define the interpretation below.

$$
\begin{array}{l}
[\![(\mathtt{p}, \mathtt{g})]\!]^{(\rho, \mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}} \stackrel{\textbf{def}}{=} \lambda\Psi, \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \wedge (\mathbb{S}.\mathbb{F}(\mathsf{if}) = \mathtt{ff}) \wedge \exists n.\ \mathsf{WFST}(n, \rho, \mathtt{g}, \mathbb{S}, \mathcal{D}, \Psi) \\
\mathsf{WFST}(0, \rho, \mathtt{g}, \mathbb{S}, \mathcal{D}, \Psi) \stackrel{\textbf{def}}{=} \\
\quad \forall \mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \exists \mathsf{ra}, \pi.\ (\mathsf{ra} = \mathsf{Ret\_Addr}(\mathbb{S})) \wedge (\mathsf{ra}, \pi) \in \Psi \wedge [\![\pi]\!]_{\mathcal{D}}\ \Psi\ \mathsf{next}_{\mathsf{ret}}(\mathbb{S}') \\
\mathsf{WFST}(n+1, \rho, \mathtt{g}, \mathbb{S}, \mathcal{D}, \Psi) \stackrel{\textbf{def}}{=} \\
\quad \forall \mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \exists \mathsf{ra}, \mathtt{p}', \mathtt{g}', \mathbb{S}''.\ (\mathsf{ra} = \mathsf{Ret\_Addr}(\mathbb{S})) \wedge (\mathsf{ra}, \langle \rho, \mathcal{L}_{\mathrm{SCAP}}, (\mathtt{p}', \mathtt{g}') \rangle) \in \Psi \\
\qquad\qquad \wedge \mathbb{S}'' = \mathsf{next}_{\mathsf{ret}}(\mathbb{S}) \wedge \mathtt{p}'\ \mathbb{S}'' \wedge \mathsf{WFST}(n, \rho, \mathtt{g}', \mathbb{S}'', \mathcal{D}, \Psi)
\end{array}
$$

The interpretation $[\![\_]\!]^{(\rho, \mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}}$ maps $(\mathtt{p}, \mathtt{g})$ to an assertion a in OCAP-x86. It takes the id ρ assigned to SCAP-x86 and a dictionary $\mathcal{D}$ as open parameters. $\mathcal{D}$ is used to interpret foreign modules that will interact with modules verified in SCAP-x86; it is decided at

$$(\textit{World})\ \ \mathbb{W} ::= (\mathbb{C}, \mathbb{S}, \mathtt{pc}, \mathtt{t}, \mathsf{T}, \mathsf{Q_r}, \mathsf{B})$$
$$(\textit{Instr})\ \ \iota ::= \ldots \mid \mathsf{scheduler} \mid \mathsf{block} \mid \mathsf{unblock} \mid \ldots$$

Fig. 10. The AIM-x86 Machine

the time of linking. The invariant about interrupts is added back, although it is omitted in p. For any p and g, we have $\forall \Psi, \mathbb{S}.\ [\![(\mathtt{p}, \mathtt{g})]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}}\ \Psi\ \mathbb{S} \rightarrow \neg \mathsf{enable\_itr}(\mathbb{S})$. The predicate WFST says that, starting from $\mathbb{S}$, every time a function returns (*i.e.,* its g is fulfilled), the return address is a valid code pointer in Ψ and its specification is satisfied after ret. Here $\mathsf{Ret\_Addr}(\mathbb{S})$ gets the return address saved on the stack of $\mathbb{S}$.

The embedding of SCAP-x86 into OCAP-x86 is sound. The soundness theorem is given below. Given a program module $\mathbb{C}$ verified in SCAP-x86, we can apply the theorem to convert it into a verified package in OCAP-x86. When the code at the Level S in Fig. 2 is verified, (*i.e.,* when $\psi \vdash \mathbb{C} : \psi'$ is derived in SCAP-x86), we do not need any knowledge of Level C and the foreign logics in which Level C is verified.

Theorem 2 (Soundness of the Embedding of SCAP-x86).
Suppose ρ is the id assigned to SCAP-x86. For all $\mathcal{D}$, let $\mathcal{D}' = \mathcal{D}\{\rho \rightsquigarrow (\mathcal{L}_{\text{SCAP}}, [\![\_]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}})\}$. If $\psi \vdash \mathbb{C} : \psi'$ in SCAP-x86, we have $\mathcal{D}'; \llcorner \psi \lrcorner_\rho \vdash \mathbb{C} : \llcorner \psi' \lrcorner_\rho$ in OCAP-x86, where $\llcorner \psi \lrcorner_\rho \stackrel{\text{def}}{=} \{(\mathtt{f}, (\rho, \mathcal{L}_{\text{SCAP}}, \theta)) \mid \psi(\mathtt{f}) = \theta\}$.

6 Verifying Synchronization Primitives and Interrupt Handlers

The code at Level C implements synchronization primitives and a timer interrupt handler. It handles interrupts explicitly and is executed by preemptive threads. It also calls the thread primitives at Level S, therefore thread queues need to be well-formed in memory. However, since all accesses to thread queues are made through these thread primitives, thread queues would always be well-formed if we treat thread primitives as atomic operations at this level. Thus we can hide this invariant from specifications, as we hide interrupts in SCAP-x86. Based on this idea, we develop an abstract interrupt machine (AIM-x86), which treats thread primitives as atomic instructions. Representations of queues and TCBs (*e.g.,* $\mathsf{embed\_RdyQ}(\mathsf{T}, \mathsf{Q})$ and $\mathsf{embed\_curr\_TCB}(\mathtt{t}, \mathtt{tcb})$) are abstracted away. Abstract representations in Fig. 8 are used instead.

Figure 10 shows the AIM-x86 abstract machine, a variation of our AIM machine [7]. We extend the x86 world with the thread ID of the current thread, a mapping of thread IDs to their TCBs, a ready queue and a set of block queues, which are all defined in Fig. 8. Then the three primitive instructions are added. The rest part of the machine are the same as in Fig. 3. Below we show the operational semantics for scheduler:

$$\frac{\begin{array}{c}\mathbb{C}(\mathtt{pc}) = \mathsf{scheduler} \quad \mathbb{F}(\mathsf{if}) = \mathbb{F}'(\mathsf{if}) = \mathtt{ff} \quad \mathsf{isr} = \mathtt{ff} \quad \mathsf{T}(\mathtt{t}) = (\mathtt{s}, \_, \_) \quad \mathbb{R}(\mathsf{sp}) = \mathtt{l}{+}2 \\ \mathsf{T}' = \mathsf{T}\{\mathtt{t} \rightsquigarrow (\mathtt{s}, \mathtt{l}, \mathtt{pc}{+}1)\} \quad \mathsf{Q_r} \cup \{\mathtt{t}\} = \mathsf{Q}'_\mathsf{r} \cup \{\mathtt{t}'\} \quad \mathsf{T}(\mathtt{t}') = (\_, \mathtt{l}', \mathtt{pc}') \quad \mathbb{R}'(\mathsf{sp}) = \mathtt{l}'{+}2 \\ \mathbb{M} = \mathbb{M}_1 \uplus \mathbb{M}_2 \quad (\mathtt{l} \overset{2}{\mapsto} \_ * \mathtt{l}{-}2 \overset{2}{\mapsto} \_ * \cdots * \mathtt{l}{-}\mathsf{sch\_stk} \overset{2}{\mapsto} \_)\ \mathbb{M}_2 \\ \mathbb{M}' = \mathbb{M}_1 \uplus \mathbb{M}'_2 \quad (\mathtt{l} \overset{2}{\mapsto} \_ * \mathtt{l}{-}2 \overset{2}{\mapsto} \_ * \cdots * \mathtt{l}{-}\mathsf{sch\_stk} \overset{2}{\mapsto} \_)\ \mathbb{M}'_2\end{array}}{(\mathbb{C}, (\mathbb{M}, \mathbb{R}, \mathbb{F}, \mathsf{isr}), \mathtt{pc}, \mathtt{t}, \mathsf{T}, \mathsf{Q_r}, \mathsf{B}) \longmapsto (\mathbb{C}, (\mathbb{M}', \mathbb{R}', \mathbb{F}', \mathsf{isr}), \mathtt{pc}', \mathtt{t}', \mathsf{T}', \mathsf{Q}'_\mathsf{r}, \mathsf{B})}\ (\text{SCH})$$

Before executing scheduler, the interrupt needs to be disabled. The scheduler saves sp (actually sp-2) and the return address (pc$+1$) into the TCB of the current thread. A

thread $\mathtt{t}'$ is picked to run and the control is transferred to $\mathtt{pc}'$, which is loaded from the TCB of $\mathtt{t}'$. The memory $\mathbb{M}$ can be split into $\mathbb{M}_1$ and $\mathbb{M}_2$. $\mathbb{M}_2$ is the stack used by scheduler. $\mathbb{M}_1$ is untouched by scheduler. We can see the semantics is consistent with sch_grt in Fig. 9, modulo the fact that the SCH rule specifies states before calling scheduler and after the return of it, while sch_grt specifies states after the call and before the return. Since pc's in thread queues are not accessible by normal instructions, they are not first-class data and we do not need to guarantee their validity in our program logic, which is challenging in Hoare-style reasoning [13].

Domain-Specific Logics for Level C. We have presented a program logic for AIM [7] in our previous work. Here we use variations of the logic to verify our code. These variations are not developed for the extended instruction sets in the abstract AIM-x86 machine; instead, scheduler, block and unblock are treated as aliases for the real x86 instructions "call `scheduler`", "call `block`" and "call `unblock`". Similar to the sequential subset of x86 in Fig. 6, AIM-x86 is simply a conceptual machine model only for the design of our domain-specific program logics; it is never implemented.

We use customized program logics, SCAP-Rdy, SCAP-Idle and SCAP-Itr, to verify normal threads, the idle thread and interrupt handlers respectively. In SCAP-Rdy, we exclude the inference rules for "iret" and "eoi", which are supposed to be used only in interrupt handlers. The set of SCAP-Idle rules is a strict subset of SCAP-Rdy excluding the rule for "block", so that we can guarantee there is at least one thread in the ready queue when the running thread is blocked. SCAP-Itr has rules for "iret" and "eoi". In the interpretation for SCAP-Itr, we distinguish the outermost level function (the interrupt handler) and functions called by the interrupt handler, since the former returns using "iret". The rules for the common instructions supported in the three logics are the same, therefore normal functions need to be verified only once and can be reused.

Embedding in OCAP-x86. We need to define interpretations for the logics to embed them in OCAP-x86. Since the well-formedness of thread queues (*e.g.,* $\mathsf{embed\_RdyQ}(\mathsf{T},\mathsf{Q})$) is assumed as invariants in the logics and is unspecified in program specifications, we need to add it back in our interpretation so that the preconditions of thread primitives at Level S (*e.g.,* sch_pre in Fig. 9) are satisfied when they are called from Level C.

The soundness of the embedding is similar to Theorem 2, which says inference rules in the customized logics can be viewed as lemmas in the OCAP-x86 frame work based on the interpretation. The following lemma says the rule for scheduler in SCAP-Rdy is sound, given any specification of scheduler compatible with $(\mathsf{sch\_pre},\mathsf{sch\_grt})$.

Lemma 3 (Interfacing with scheduler). Suppose ρ_r is the id for SCAP-Rdy. For all $\mathcal{D}$, let $\mathcal{D}' = \mathcal{D}\{\rho \rightsquigarrow (\mathcal{L}_{\text{SCAP}}, [\![\_]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}}), \rho_r \rightsquigarrow (\mathcal{L}_{\text{RDY}}, [\![\_]\!]^{\rho_r}_{\text{RDY}})\}$. If $\Psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f}:$ scheduler in SCAP-Rdy, $\Psi = \llcorner\Psi\lrcorner_{\rho_r} \cup \{(\mathtt{scheduler}, (\rho, \mathcal{L}_{\text{SCAP}}, (\mathtt{p}_s,\mathtt{g}_s)))\}$, and $(\mathsf{sch\_pre},\mathsf{sch\_grt}) \Rightarrow (\mathtt{p}_s,\mathtt{g}_s)$, we have $\mathcal{D}', \Psi \vdash \{[\![\,(\mathtt{p},\mathtt{g})\,]\!]^{\rho_r}_{\text{RDY}}\}$ call `scheduler` in OCAP-x86.

Here $[\![\_]\!]^{\rho_r}_{\text{RDY}}$ is the interpretation for SCAP-Rdy, and $(\mathtt{p},\mathtt{g}) \Rightarrow (\mathtt{p}',\mathtt{g}')$ is defined as:

$$(\forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{p}'\ \mathbb{S}) \wedge (\forall \mathbb{S}, \mathbb{S}'.\ \mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}')\,.$$

We do not show the details of the program logics and their interpretations. Specifications for primitives at Level C are similar to those for the AIM implementations [7]. Interested readers can refer to the Coq implementation [8] for more details.

Component	# of lines
Basic Utility Definitions & Lemmas	2,766
Machine, Opr. Semantics & Lemmas	3,269
Separation Logic, Lemmas & Tactics	6,340
OCAP-x86 & Soundness	1,711
SCAP-x86 & Soundness	1,357
Thread Queues & Lemmas	1,199
SCAP-Rdy, SCAP-Idle & SCAP-Itr	26,347

Component	# of lines
Assembly Code at Level S	292*
enQueue/deQueue	4,838
scheduler, block, unblock	7,107
Assembly Code at Level C	411*
Timer Handler	2,344
`yield` & `sleep`	7,364
locks: `acq_h` & `rel_h`	10,838
cond. var.: `wait_m` & `signal_m`	5,440

* They are the Coq source files containing the encoding of the assembly code. The real assembly code in these two files is around 300 lines.

Fig. 11. The Verified Package in Coq

7 Implementations in Coq

In our Coq implementations, we use a simplified version of OCAP-x86, which instantiates the full-version OCAP-x86 with a priori knowledge of the four domain-specific logics (SCAP-x86, SCAP-Rdy, SCAP-Idle and SCAP-Itr). The soundness of the framework and the embedding of the four logics have been formally proved in Coq. We have also verified most of the modules shown in Fig. 2, which have around 300 lines of x86 assembly code. The whole Coq implementation has around 82,000 lines, including 1165 definitions and 1848 lemmas and theorems. Figure 11 gives a break down of the number of lines for various components.

The implementation has taken many man-months, including the implementation of basic facilities such as lemmas and tactics for partial mappings, queues, and separation logic assertions. The lesson we learn is that writing proofs is very similar to developing large-scale software systems — many software-engineering principles would be equally helpful for proof development; especially a careful design and a proper infrastructure is crucial to the success. We started to prove the main theorems without first developing a proper set of lemmas and tactics. The proofs done at the beginning used only the most primitive tactics in Coq. Auxiliary lemmas were proved on the fly and some were proved multiple times by different team members. The early stage was very painful and some of our members were so frustrated by the daunting amount of work. After one of us ported a set of lemmas and tactics for separation logic from a different project [12], we were surprised to see how the proof was expedited. The tactics manipulating separation logic assertions, especially the ones that reorder sub-terms of separating conjunctions, have greatly simplified the reasoning about memory.

Another observation is that verifying both the library (*e.g.,* Level S primitives) and its clients (*e.g.,* Level C code) is an effective way to validate specifications. We have found some of our initial specifications for Level S code are too strong or too weak to be used by Level C. Also, to avoid redoing the whole proof after fixing the specifications, it is helpful to decompose big lemmas into smaller ones with clear interfaces.

The size of our proof scripts is huge, comparing with the size of the assembly code. This is caused in part by the duplicate proof of the same lemmas by different team members. Another reason is the lack of proper design and abstraction: when an instruction is seen a second time in the code, we simply copy and past the previous proof and do

some minor changes. The proof is actually developed very quickly after introducing the tactics for separation logic. For instance, the 5440 lines Coq code verifying condition variables is done by one of the authors in two days. We believe the size of proof scripts can be greatly reduced with more careful abstractions and more reuse of lemmas.

8 Related Work and Conclusions

Bevier [1] verified Kit, an OS kernel implemented in machine code, using the Boyer-Moore logic. Gargano *et al.* [10] showed a framework to construct a verified OS kernel in the Verisoft project. Both kernels support process scheduling and interrupts (and more features). Their interrupt handlers are part of the kernel, but the kernels are sequential and cannot be interrupted. There are no kernel level threads either. In both cases, it is not clear how to verify the concurrent higher-level code (processes in Kit [1] or CVMs in the Verisoft project [10]) and link it with the verified kernel as a whole system. The CLI stack project [2] verified sub-systems at different abstraction levels and composed them as a whole verified system. Interactions between different levels were formalized and verified. The verification, however, was based on operational semantics and may not be scalable to handle concurrency and higher-order code pointers. Ni *et al.* [14] verified a non-preemptive thread library in XCAP [13], which treats `pc` stored in TCBs as first-class continuations. The library, however, is not linked with verified threads. Elphinstone *et al.* [5] proposed to implement a prototype of kernel in Haskell, and to verify that the C implementation satisfies the Haskell specification.

In this paper, we present an application of our open-framework-based methodology for system verification. We verify thread implementations, synchronization primitives and interrupt handlers using domain-specific logics and link them in the open framework to get a verified whole system. The work is based on our previous work addressing various theoretical problems, *i.e.,* the OCAP framework [6], the SCAP logic [9] and the AIM machine and logic for preemptive threads and interrupts [7]. In our previous work on OCAP [6], we showed how to link a verified scheduler with non-preemptive threads. That was a proof-of-concept example, which was not developed for real machine architecture and did not support interrupts.

There might be alternative solutions to some of the problems shown in Sect. 2. For instance, using separation logic's hypothetical frame rules [16], it is also possible to hide the concrete representations of thread queues for Level C. However, it is not clear how to support first-class code pointers and to ban certain instructions in specific scenarios in separation logic. We manage to address all these issues in a single framework. Also, our methodology is general enough to support domain-specific logics with different specification languages, *e.g.,* type systems and Hoare logics [6].

On the other hand, it is important to note that our methodology and framework do not magically solve interoperability problems for arbitrary domain-specific logics. Embedding foreign logics into the framework and letting them interact with each other involve non-trivial theoretical and engineering problems. For instance, invariants enforced in different logics need to be recognized and be properly encoded in the foundational logic. The encoding may also require mappings of program states from higher abstraction levels to lower levels. The problems may differ in specific applications. We would like to apply our methodology to larger applications (*e.g.,* verifying our 1300-line OS kernel) to further test its applicability.

Acknowledgment

We thank anonymous referees for their suggestions and comments. Wei Wang, Haibo Wang, and Xi Wang helped prove some of the lemmas in our Coq implementation. Xinyu Feng and Zhong Shao are supported in part by gift from Microsoft and NSF grant CCR-0524545. Yu Guo is supported in part by grants from National Natural Science Foundation of China (under grants No. 60673126 and No. 90718026) and Intel China Research Center. Yuan Dong is supported in part by National Natural Science Foundation of China (under grant No. 60573017), Hi-Tech Research And Development Program Of China (under grant No. 2008AA01Z102), China Scholarship Council, and Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList). Any opinions, findings, and contributions in this document are those of the authors and do not reflect the views of these agencies.

References

[1] W. R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, 1989.

[2] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[3] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *PLDI'07*, pages 66–77, June 2007.

[4] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.1.

[5] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[6] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *TLDI'07*, pages 67–78, January 2007.

[7] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI'08*, page to appear, June 2008.

[8] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems, extended version and Coq implementations. `http://flint.cs.yale.edu/flint/publications/itrimp.html`, 2008.

[9] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, June 2006.

[10] M. Gargano, M. A. Hillebrand, D. Leinenbach, and W. J. Paul. On the correctness of operating system kernels. In *TPHOLs'05*, 2005.

[11] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Programming Languages and Systems*, 5(4):596–619, 1983.

[12] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *PLDI'07*, pages 468–479, June 2007.

[13] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symp. on Principles of Prog. Lang.*, pages 320–333, 2006.

[14] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs'07*, pages 189–206, 2007.

[15] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04*, volume 3170 of *LNCS*, pages 49–67, 2004.

[16] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, Jan. 2004.

[17] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*, 1993.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.