Modular Verification of Multi-Threaded Assembly Code

Xinyu Feng and Zhong Shao
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{feng, shao}@cs.yale.edu

Abstract

Concurrency, as a useful feature of many modern programming languages and systems, is generally hard to reason about. Verification of concurrent assembly code is even harder, largely due to the lack of abstraction at the low level. In this paper, we present a certified programming framework for verifying concurrent assembly code with unbounded dynamic thread creation. We apply the rely-guarantee methodology at the assembly level and show how to specify the semantics of thread “fork” with argument passing. In particular, we allow threads have different assumptions and guarantees at different stages of lifetime so that they can coexist with the dynamic thread environment resulted from thread creation and termination. Our thread model is close to the real machine model and can support realistic multi-threaded programs.

1 Introduction

Proof-Carrying Code (PCC) [13] is a general framework for verifying the safety properties of machine-language programs. It allows a code producer to provide a machine language program to a host along with a formal proof of its safety. The proof can be mechanically checked by the host and the producer need not be trusted because a valid proof is a dependable certificate of safety. Although many verification frameworks have been proposed to generate PCC packages [13, 12, 2, 10, 21, 4, 22], most of them only support sequential programs, except for the work on CCAP [22], which applies the rely-guarantee methodology [11] to specify and reason about general properties of concurrent programs at the assembly level.

However, the support of multi-threaded programming in CCAP is insufficient in that CCAP, like existing work on verification of high-level concurrent programs, only supports properly nested concurrent code in the form of \( \text{cobegin } P_1 \parallel \ldots \parallel P_n \text{ coend} \). This seriously restricts CCAP’s application, given the fact that thread management, including dynamic thread creation, is a widely supported feature in operating systems.

Dynamic thread creation (and termination) implies a changing thread environment, \( i.e., \) the collection of all live threads in the system other than the thread under concern. The concept of environment is essential for compositional concurrent program verification. With a dynamic environment, many issues arise: we want to ensure that the newly created thread will not interfere with existing live threads, yet we do not want to enforce the non-interference property for threads with no overlap in their lifetime (\( e.g., \) one thread created after the termination of another).

Modularity is another challenge to make the PCC technique scale. Existing work on rely-guarantee methodology supports thread modularity, \( i.e., \) different threads can be verified separately without looking into other threads’ code. However, they do not have a good support of code reuse: if a procedure is called in more than one threads, it must be verified multiple times using different specifications, one for each calling thread. Ideally, we want a procedure to be specified and verified once, then the PCC package for this procedure can be reused for different threads.

In this paper, we extend the work on CCAP and propose a language for certified multi-threaded assembly programming (CMAP). CMAP is based on a realistic abstract machine which supports dynamic thread creation with argument passing (the “fork” operation). Thread “join” can also be implemented in our language using synchronization.

Our work makes the following new contributions:

- The “fork/join” thread model is more general than “cobegin/coend” in that it supports unbounded dynamic thread creation which poses new challenges for verification. To our knowledge, our work is the first to apply the rely-guarantee method to verify assembly code with unbounded dynamic thread creation. In particular, we unify the concepts of a thread’s assumption/guarantee and its environment’s guarantee/assumption, and allow a thread to change its assumption/guarantee to track the composition of its dynamically changing environment.
In addition to thread modularity, our work has good support of code (and verification) reuse. Code segments can be specified and verified once but used in multiple threads.

We solve some practical issues like argument passing at thread creation, multiple “incarnation” of the same piece of thread code, etc. These issues are important for realistic multi-threaded programming, but have never been discussed in existing work.

In the rest of this paper, we first give an overview of the rely-guarantee-based reasoning and a detailed explanation of the key challenges in verifying multi-threaded assembly code (Section 2). We then present our work on CMAP with formal semantics (Section 3). We use a few examples to illustrate CMAP-based program verification in Section 4. Finally, we discuss related work and then conclude.

2 Background and Challenges

2.1 Rely-Guarantee-Based Reasoning

The rely-guarantee (R-G) proof method [11] is one of the best-studied approaches to the compositional verification of shared-memory concurrent program. Under the R-G paradigm, every thread is associated with a pair \((A, G)\), with the meaning that if the environment \(i.e.,\) the collection of all of the rest threads) satisfies the assumption \(A\), the thread will meet its guarantee \(G\) to the environment. In the shared-memory model, the assumption \(A\) of a thread describes what atomic transitions may be performed by other threads, and the guarantee \(G\) of a thread must hold on every atomic transition performed by the thread itself. They are typically modeled as predicates on a pair of states, which are often called actions.

For instance, in Figure 1 we have two interleaving threads \(T_1\) and \(T_2\). \(T_1\)’s assumption \(A_1\) adds constraints on the transition \((S_0, S_1)\) made by the environment \((T_2\) in this case), while \(G_1\) describes the transition \((S_1, S_2)\), assuming the environment’s transition satisfies \(A_1\). Similarly, \(A_2\) describes \((S_1, S_2)\) and \(G_2\) describes \((S_0, S_1)\).

Figure 1. Rely-Guarantee-Based Reasoning

Figure 2. R-G in a non-preemptive setting

We need two steps to reason about a concurrent program consisting of \(T_1\) and \(T_2\). First, we check there is no interference between threads, that is, each thread’s assumption can indeed be satisfied by its environments. In our example, the non-interference is satisfied as long as \(G_1 \Rightarrow A_2\) (a shorthand for \(\forall S, S'. G_1(S, S') \Rightarrow A_2(S, S')\)), and \(G_2 \Rightarrow A_1\). Second, we check that \(T_1\) and \(T_2\) do not lie, that is, they do satisfy their guarantee as long as their assumption is satisfied. As we can see, the first step only uses the specification of each thread, while the second step can be carried out independently without looking at other threads’ code. This is how the R-G paradigm achieves thread-modularity.

2.2 R-G in Non-Preemptive Thread Model

CMAP adopts a non-preemptive thread model, in which threads yield control voluntarily with a \texttt{yield} instruction, as shown in Figure 2. The preemptive model can be regarded as a special case of the non-preemptive one, in which an explicit \texttt{yield} is used at every program point. Also, on real machines, programs might run in both preemptive settings and non-preemptive settings: preemption is usually implemented using interrupts; a program can disable the interrupt to get into the non-preemptive setting.

An “atomic” transition in a non-preemptive setting then corresponds to a sequence of instructions between two \texttt{yield}. For instance, in Figure 2 the state pair \((S_2, S_2')\) corresponds to an atomic transition of thread \(T_1\). A difficulty in modeling concurrency in such a setting is that the effect of an “atomic” transition cannot be completely captured until reaching the end. For example, in Figure 2, the transition \((S_1, S_1')\) should satisfy \(G_2\). But when we reach the intermediate state \(S\), we have no idea of what the whole transition \(i.e.,\) \((S_1, S_1')\) will be. At this point, neither \((S_1, S)\) nor \((S, S_1')\) need satisfy \(G_2\). Instead, it may rely on its remainder commands (the commands between \texttt{comm} and \texttt{yield}, including \texttt{comm}) to complete an adequate state transition. In CCAP [22], a “local guarantee” \(g\) is introduced for every program point to capture further state changes that must be made by the following commands before it is safe for the current thread to yield control. For instance, the local guarantee \(g\) attached to \texttt{comm} describes the transition \((S, S_1')\).
2.3 Challenges for Dynamic Thread Creation

When using the rely-guarantee method to verify multi-threaded programs, it is natural to assign an assumption and guarantee to each thread, which is the basic language construct to support concurrency. The non-interference can be enforced by checking all these assumptions and guarantees, as is done in [22] and [7]. However, this simple approach no longer works when we support dynamic thread creation and termination. Thread creation and termination imply that threads have different lifetime, and for each thread the composition of its “environment” is dynamic during its lifetime. To see this, we first have a look at the high-level pseudo code (in C-like syntax) shown in Figure 3, in which the main thread creates two child threads to compute the GCD (greatest common divisor) of \( \alpha \) and \( \beta \).

The main thread initiates the variable \( a \) and \( b \), creates two child threads using the same copy of gcd function with different arguments, waits until the termination of them, and finally does some post processing of \( a \) and \( b \). Note here we do not have thread “join”, instead we use flags to implement the synchronization. Also assume each statement is an atomic operation and threads can be interrupted between any two statements.

To verify the program, we must address the following issues:

1. As shown in Figure 4 (a) (time goes downwards), when doing data initiation (A-B) and post processing (C-D), the main thread need assume that no other threads in the environment (say, \( T_3 \)) can change the value of \( a \), \( b \) and flags.\(^1\) However, the composition of the main thread’s environment changes after it creates child threads, who will collaborate with it. The assumption used at A-B and C-D is no longer appropriate for this new environment. How can we specify the main thread to support such a dynamic thread environment?

One possible approach is that the main thread relaxes its assumption to make exceptions for its child threads. However, it is hard to specify the parent-child relationship. Another approach is to use something (e.g., the program counter) in the assumption and guarantee to indicate the phase of computation. At the phase B-C, the assumption is relaxed. This approach is not satisfactory either, because the program structure of the main thread has to be exposed to the specification of the child threads. If the main thread is re-written, the specification of the child threads has to be revised accordingly.

2. The “fork” and “join” are not properly nested in the program, although it is trivial to switch these two “while” loops to make it be. Our logic should be general enough to support the improperly nested program structures, as shown in Figure 4 (b).

3. Since two child threads are created using the function gcd, we must make sure that there is no interference between these two incarnations. This may require the guarantee assigned to gcd implies its own assumption. However, this requirement is too strong for cases when only one thread is created (from the code), e.g., the main thread. How can we distinguish these two cases and treat them differently?

4. Another issue introduced by dynamic thread creation, but not shown in this program, is that the lifetime of some threads may never overlap. In the case shown in Figure 4 (c), lifetime of \( T_2 \) and \( T_3 \) do not overlap and we should not enforce the non-interference between them. Again, how can we specify and check the interleaving of threads?

In the next section, we’ll show how these issues are solved in the development of CMAP.

\(^1\)One may argue that they are local variables and invisible to other threads, but at the raw machine level we do not have such abstractions of local and global variables. All we have is shared flat memory.
then (State) S ::= (M, R)
(Memory) M ::= {t~ w}+
(Register) r ::= r0 | r1 | ... | rt | ra
(CdHeap) C ::= {c ~ I}+
(Labels) f, l ::= n
(WordVal) w ::= n
(TEntries) T ::= {t ~ II}+
(TQueue) Q ::= {t ~ (R, I)}+
(THandles) h ::= n
(ThrID) t ::= n
(ThrArg) a ::= n
(InstrSeq) I ::= c; I | jdl f | exit

**Figure 5. The Abstract Machine**

### 3 CMAP

The language CMAP is based on an “untyped” low-level abstract machine supporting multi-threaded programs with dynamic thread creation and argument passing. The “type” system of CMAP uses the calculus of inductive constructions (CiC) [15] to support essentially reasoning in higher-order predicate logic.

#### 3.1 The Abstract Machine

A CMAP program (corresponding to a complete machine state) consists of a state S (which consists of the shared memory M and the register file R), a dynamic thread queue Q, two shared code heaps C (for basic code blocks) and T (for thread entries), and the current instruction sequence I of the currently executing thread. Here I plays the role of program counter. The register file R consists of 16 general purpose registers (r0 - r15) and two special registers (i.e., rt and ra) holding the current thread id and the thread argument.

For simplicity, we just model the queue of ready threads, which is the dynamic thread queue Q. The dynamic thread id t is a non-zero natural number randomly generated at run time. Each entry in Q is an execution context of a thread, including the snapshot of the register file and the program point where the thread will resume its execution. Since we support multiple “incarnation” of one copy of code, it is necessary to have Q in addition to T. We also call each thread entry in T a “static thread”.

The instruction set of CMAP just contains the most basic and common assembly instructions. It also includes primitives fork, exit and yield to support multi-threaded programming (these primitives can be viewed as system calls to the thread library). Readers who are eager to see a CMAP program can take a quick look at Figure 10 and 11 (ignore the annotations for the time being), which are the CMAP implementation of the GCD program shown in Figure 3. Note we do not have join; the thread join can be implemented by means of synchronization, as shown in the example.

Execution of CMAP programs is modeled as transitions from one program to another, i.e., P ⊢ P’. Figure 6 defines the program transition function as the operational semantics.

```
if c = then Next(c, (M, R)) =
  add r_d, r_s, r_t (M, R{r_d ~ R(r_s) + R(r_t)})
  sub r_d, r_s, r_t (M, R{r_d ~ R(r_s) - R(r_t)})
  movi r_d, w (M, R{r_d ~ w})
  ld r_d, r_s (w) (M{R(r_d) + w} ~ R(r_s), R)
  where (R(r_d) + w) ∈ dom(M)
```

**Figure 7. Auxiliary State Update Macro**

### 3.2 The Meta-Logic

To encode the specification and proofs, we use the calculus of inductive constructions (CiC) [18, 15], which is an extension of the calculus of constructions (CC) [3] with inductive definitions. CC corresponds to Church’s higher-order predicate logic via the Curry-Howard isomorphism.

CiC has been shown strongly normalizing [19], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [18], which we use to implement the results presented in this paper.
In the remainder of this paper, we will mainly use
the more familiar mathematical and logical notations, instead
of strict CiC or Coq representation. No knowledge of CiC
and Coq is required to understand them.

3.3 Program Specifications

Verification constructs of CMAP are defined in Figure 8.
The program specification $\Phi$ consists of a global invariant
($Inv$), the static threads specification $\Delta$, and the code heap
specification $\Psi$. The global invariant $Inv$ is a programmer
specified predicate, which implies a safety property of con-
cern. It must hold throughout the execution of the program.
The static threads specification $\Delta$ contains specifications $\theta$
for each static thread in $T$. Each $\theta$ consists of an assertion $p$
that describes the requirement of states at which this thread is
invoked, and the assumption $A$ and guarantee $G$
of this thread with regard to the environment at its creation
time. The assertions (e.g., $p$ and $Inv$) are CiC terms with
type $State \rightarrow Prop$, i.e., predicates over states. Assumptions
and guarantees (e.g., $A$, $G$ and $g$) are CiC terms with type
$Reg \rightarrow Mem \rightarrow Mem \rightarrow Prop$, which means predicates
over a register file and two copy of memory (instead of predicates
over a pair of states, as introduced in Section 2.1. We will
explain this in the following paragraphs).

The code heap specification $\Psi$ assigns a quadruple
($p, g, A, G$) to each instruction sequence. The assertion $p$
describes expected states to execute the code sequence. The
local guarantee $g$, as introduced in Section 2.2, describes
a valid state transition – it is safe for the current thread to
yield control only after making a state transition allowed by
$g$. When a thread executes this instruction sequence, we al-
low it to use the $A$ and $G$ specified here, rather than those
in $\theta$, as its assumption and guarantee. Here the $A$ and $G$
reflect the knowledge of the dynamic thread environment at
the time when the instruction sequence is executed. In this

way threads can have different assumptions and guarantees
during their lifetime.

**Thread-Local State in A and G.** Since the register file $R$
contains the thread-local data, the $yield$ instruction changes
$S$ after context switch, which makes the meaning of $A$ and
$G$ involved. In Figure 2 the state $S_0$ becomes different
with $S_1$, similarly for $S'_1$ and $S_2$, and $S'_2$ and $S_3$. How-
ever, we note that the only difference between the states
before and after $yield$ is the register file $R$, while assump-
tions and guarantees only talk about the shared memory $M$.
If we formulate $A$ and $G$ as predicates over a pair of memory,
the gap disappears. On the other hand, we also want $A$ and $G$
be polymorphic over the register file. For instance, we allow multiple invocation of one static threads
with different arguments in $ra$. The assumptions and guar-
antees of different dynamic copies may depend on their ar-
guments. Therefore $A$ and $G$ (similarly for $g$) have type
$Reg \rightarrow Mem \rightarrow Mem \rightarrow Prop$. Moreover, we require the
global invariants be independent of the register file, i.e.,
$\forall R, R'. \forall M. Inv (M, R) \Rightarrow Inv (M, R')$.

**Specification of Dynamic Thread Queue.** We also intro-
duce in CMAP the specification $\Theta$ of active threads in the
thread queue Q. Within Θ, each triple (p, A, G) describes a
dynamic thread at its yield point (or at the beginning). The
assertion p gives the constraint of states when the thread
gets control back to execute remaining instructions. The
assumption and guarantee used by the thread at the yield
point are given by A and G. Since we allow threads to have
different assumptions and guarantees at different stages of
lifetime, A and G do not have to be the same with those in
the specification θ of the corresponding static thread.

3.4 Inference Rules

We use the following judgement forms to define the
inference rules:

\[ \Phi; \Theta; (p, g, A, G) \vdash p \] (well-formed program)
\[ \Phi; \Theta; (g, S) \vdash Q \] (well-formed dynamic threads)
\[ \Phi \vdash T \] (well-formed static threads)
\[ \Phi \vdash C \] (well-formed code heap)
\[ \Phi; (p, g, A, G) \vdash I \] (well-formed instr. sequence)

Well-formed Program. The PROG rule shows the invari-
ants that need to be maintained during program transitions.

\[ \begin{align*}
(Inv, \Delta, \Psi) \equiv & \Phi, (M, \mathcal{R}) = S \quad t = \mathcal{R}(rt) \\
\Phi \vdash T & \\
\Phi \vdash C & \\
\Phi; (g, S) \vdash Q \\
\Phi; \Theta; (p, A, G) \vdash I \\
\Phi; (p, g, A, G) \vdash (S, Q, T, C, I) &
\end{align*} \]

(PROG)

The well-formedness of a program is judged with respect
to the program specification Φ, the dynamic thread spec-
fication Θ, and the specification of the current executing
thread (p, g, A, G). Compared with the triples in Θ, we
need the local guarantee g here to specify the valid trans-
ition the current thread must make before it yields control.

In the first line, we give the composition of the program
specification, the current state and the current thread id.
Here we use a pattern match representation, which will be
used in the following part of this paper.

We require the code, including the thread entry points T
and the code heap C, always be well-formed with respect
to the program specification. Since Φ, T and C do not
change during program transitions, the check for the first
two premises in line 2 can be done once for all.

The next premise shows the constraints on the current
state S: it must satisfy both the global invariant Inv and the
assertion p of the current thread. The last premise in line 2
essentially requires it be safe for the current thread to exe-
cute the remainder instruction sequence I.

Premises in line 3 require the well-formedness of dy-
namic threads in Q, which is checked by the rule DTHRDS,
and the non-interference between all the live threads.

Non-Interference. The non-interference macro
NI(Θ, Q), as explained in Section 2.1, requires that
each thread be compatible with all the other. It is formally
defined as:

\[ \forall t_i, t_j \in \text{dom}(\Theta). \forall M, M'. \]
\[ t_i \neq t_j \Rightarrow \Theta(t_i), M, M' \Rightarrow A_i, R, M M', \]

where ( A_i, G_i ) = Θ(t_i), ( A_j, G_j ) = Θ(t_j), R_i = Q(t_i) and R_j = Q(t_j).

We enforce the non-interference between live threads in
Q instead of static threads in T for two reasons:

1. Some static threads may be invoked multiple times.
   As explained in section 2.3, if we enforce the non-
   interference based on the specifications of static
   threads, we must check the specifications of these
   threads in a different way (with those that will be
   activated only once) to make sure that their multiple
   instantiations are compatible. Simply enforcing the
   non-interference between dynamic threads allows us
   to avoid such a special check and treat these two kinds
   of static threads in a uniform way.

2. For threads have no overlap in their lifetime, they will
   not be in the dynamic thread queue at the same time.
   Therefore we do not have to check their interference.
   We cannot avoid such a check if it is done based on the
   specifications of static threads.

Well-formed Dynamic Threads. The rule DTHRDS en-
sures each dynamic thread in Q is in good shape with re-
spect to their specification Θ, the current program state S
and the transition g that the current thread need do before
these waiting threads can take control.

\[ (R_k, l_k) = Q(t_k), \quad (p_k, A_k, G_k) = Θ(t_k), \quad \forall t_k \in \text{dom}(Q) \]
\[ \forall M', M''. (Inv \wedge p_k)(M', R_k) \Rightarrow A_k, R, R, M' M'' \Rightarrow p_k(M'', R_k) \]
\[ (Inv, \Delta, \Psi); (p_k, G_k, A_k, G_k) \vdash l_k \]
\[ \forall M', g: \mathcal{R}, M, M' \Rightarrow p_k(M', R_k) \]

\[ (Inv, \Delta, \Psi); (g, (M, \mathcal{R})) \vdash Q \] (DTHRDS)

The first line gives the specification of each thread when
it reaches a yield point, and its execution context.
Line 2 requires that if it is safe for a thread to take control
at a state with memory M', it should also be safe to do so
after any state transition satisfying its assumption.
Line 3 requires that for each thread its remainder instruc-
tion sequence must be well formed. Note we use G_k as the
local guarantee because after yield, a thread starts a new
“atomic transition” described by its global guarantee.

The last premise describes the relationship between the
local guarantee of the current thread and the preconditions
of other threads. For any transitions starting from the cur-
rent state (M, R), as long as it satisfies g, it should be safe
for other threads to take control at the result state.
Thread Creation The FORK rule describes constraints on new thread creation, which enforces the non-interference between the new thread and existing threads.

\[
\begin{align*}
(p', A', G') &= \Delta(t) \quad A \Rightarrow A'' \quad G' \Rightarrow G \\
(A \lor G'') &= A' \quad G' \Rightarrow (G \land A'') \\
\forall(M, R), \forall M', \forall R'. (\text{Inv} \land p) (M, R) &\Rightarrow R'(t) \neq R'(rt) \\
\Rightarrow R(r) &= R'(ra) \Rightarrow g \lor M \lor M' \Rightarrow p' (M', R') \\
(Inv, \Delta, \Psi); (p, g, A, G) &\vdash \text{fork } h, r; i & \text{ (FORK)}
\end{align*}
\]

For the parent thread, the newly created child thread becomes part of its environment after “fork”. The remainder instructions of the parent thread should not interfere with this new environment. Since \(A\) also reflects the guarantee of the environment before “fork”, the guarantee of the new environment can be described by \(A \lor G'\), which takes the guarantee of the child thread into account. Similarly, \(G \land A'\) describes the assumptions of the new environment.

To verify the subsequent instruction sequence, the programmer needs to come up with a new assumption \(A''\) and guarantee \(G''\) which are compatible with the new environment.

For the child thread, its environment consists of the parent thread’s old environment and the remainder part of the parent thread itself. The assumption and guarantee of such an environment can be specified by \(G \land A'\) and \(A \lor G'\), respectively. We must make sure that the child thread is compatible with its environment.

Premises in the first two lines specify above constraints, where \((A \lor G'') \Rightarrow A'\) is the shorthand for:

\[
\begin{align*}
\forall(M, R), \forall M', \forall R'. (\text{Inv} \land p) (M, R) &\Rightarrow R'(t) \neq R'(rt) \\
\Rightarrow R(r) &= R'(ra) \Rightarrow (A \lor G'') \lor M' \lor M'' \Rightarrow A' \lor R' \lor M' \lor M'', \\
\end{align*}
\]

and \(G' \Rightarrow (G \land A'')\) for:

\[
\begin{align*}
\forall(M, R), \forall M', \forall R'. (\text{Inv} \land p) (M, R) &\Rightarrow R'(t) \neq R'(rt) \\
\Rightarrow R(r) &= R'(ra) \Rightarrow G' \lor R' \lor M' \lor M'' \Rightarrow (G \land A'') \lor R' \lor M' \lor M''.
\end{align*}
\]

The premise in line 3 says that after the current thread completes the transition described by \(g\), it should be safe for the new thread to take control. The last premise checks the well-formedness of the remainder instruction sequence. Since the FORK instruction does not change states, we need not change the precondition \(p\) and \(g\).

In most cases, the programmer can just pick \((A \lor \widehat{G'})\) and \((G \land \widehat{A'})\) as \(A''\) and \(G''\) respectively, where \(\widehat{G'}\) and \(\widehat{A'}\) are instantiations of \(G'\) and \(A'\) using the value of the child’s argument (see the example in Section 4).

The following lemmas about thread creation can be proved:

**Lemma 3.1 (Non-Interference-I)**

Suppose threads \(t'_1\) and \(t'_2\) are forked by \(t_1\) and \(t_2\) respectively. If \(t_1\) and \(t_2\) do not interfere (i.e., their specification satisfy \(NI\) defined above), and the creation of \(t'_1\) and \(t'_2\) satisfy the FORK rule, \(t'_1\) and \(t'_2\) do not interfere.

**Lemma 3.2 (Non-Interference-II)**

If threads \(t'\) and \(t''\) are forked by \(t\), and the creation of them satisfies the FORK rule, \(t'\) and \(t''\) do not interfere.

**Yielding and Termination**

\[
\begin{align*}
\forall R, \forall M, \forall R'. (\text{Inv} \land p) (M, R) &\Rightarrow A \land M \land M' \Rightarrow p (M', R) \\
\forall(M, R), (\text{Inv} \land p) (M, R) &\Rightarrow g \land R \land M \\
(Inv, \Delta, \Psi); (p, g, A, G) &\vdash \text{fork } h, r; i & \text{ (YIELD)}
\end{align*}
\]

The YIELD rule requires that it is safe for the yielding thread to take back control after any state transition satisfying the assumption \(A\). Also the current thread cannot yield until it completes the required state transition, i.e., an identity transition satisfies the local guarantee \(G\). Lastly, one must verify the remainder instruction sequence with the local guarantee reset to \(G\).

\[
\begin{align*}
(Inv, \Delta, \Psi); (p, g, A, G) &\vdash \text{exit} & \text{ (EXIT)}
\end{align*}
\]

The rule \text{EXIT} is simple: it is safe for the current thread to terminate its execution only after it finishes the required transition described by \(g\), which is an identity transition.

**Support of Modularity.** The following \text{JD} rule shows our support of modularity.

\[
\begin{align*}
(p', G', A', G') &= \Psi(e) \quad A \Rightarrow A' \quad G \Rightarrow G \\
(Inv, \Delta, \Psi); (p', g, A, G) &\vdash \text{jd } e & \text{ (JD)}
\end{align*}
\]

As mentioned before, the specification of each instruction sequence contains an assumption and guarantee. The well-formedness of the instruction sequence is verified only with regard to its own specification. It is safe for a thread to execute an instruction sequence as long as executing the instruction sequence does not require a stronger assumption than the thread’s assumption \(A\), nor does it break the guarantee \(G\) of the thread.

Interested readers can find the rest of inference rules and the soundness proof of CMAP in Appendices.

4 Experiments

4.1 Partial Correctness of A Lock-Free Algorithm

Figure 10 and 11 give the CMAP implementation of the GCD algorithm shown in Figure 3, which is a lock-free algorithm, i.e., no synchronization is required to ensure the non-interference even if atomic operations are machine instructions. To show the lock-free property, we insert \text{yield} instructions at every program point of the child thread (some \text{yield} instructions are omitted in the main thread for clarity).
id₁ ≡ a = a’  id₂ ≡ b = b’
id₃ ≡ (flag[0] = flag’[0]) ∧ (flag[1] = flag’[1])

G₀ ≡ True  Ao ≡ id₁ ∧ id₂ ∧ id₃
p ≡ id₂ ∧ (a > b ⇒ (GCD(a, b) = GCD(a’, b’))) ∧ (a ≤ b ⇒ id₂)
p’ ≡ id₃ ∧ (a < b ⇒ (GCD(a, b) = GCD(a’, b’))) ∧ (a ≥ b ⇒ id₂)

G₉ ≡ (flag[ra] = 0 ⇒ (flag[ra] = 0 ∧ p) ∨ (flag[ra] = 1 ∧ p’)) ∧
    (flag[ra] = 1 ⇒ S = S’)
    ∧ (flag[1 − ra] = flag’[1 − ra])
A₉ ≡ (flag[ra] = 0 ⇒ (flag[ra] = flag’[ra])
    ∧ (flag[0] = 0) ∨ (flag[0] = 0 ⇒ p))
G₁ ≡ G₀ ∧ (flag’[0] = 0 ⇒ (flag[0] = flag’[0]) ∧ p)
A₁ ≡ A₀ ∨ ((flag[1] = flag’[1]) ∧ (flag[0] = 0 ⇒ p)
    ∧ (flag[0] = 1 ⇒ (id₁ ∧ id₂ ∧ a’ = b’))
    ∧ (flag[1] = 1 ⇒ S = S’))
G₂ ≡ G₁ ∧ (flag’[1] = 0 ⇒ (flag[1] = flag’[1]) ∧ p)
A₂ ≡ A₁ ∨ ((flag[0] = flag’[0]) ∧ (flag[0] = 0 ⇒ p)
    ∧ (flag[1] = 1 ⇒ (id₁ ∧ id₂ ∧ a’ = b’))
    ∧ (flag[1] = 1 ⇒ S = S’))

g ≡ S = S’  p₁ ≡ a = b  p₂ ≡ GCD(a, b) = GCD(a’, b’)

Figure 9. GCD: Assertion Definitions

This also shows that our logic is general enough to simulate preemptive thread model.

To verify the partial correctness of the program, the programmer need find a global invariant and specifications for each static thread and the code heap. In Figure 9 we show definitions of assertions that are used to specify the program. For ease of reading, we named variables as shorthand for their values in memory. The primed variables represent the value of the variable after state transition. We also introduce the shorthand [x] for R[x].

The following formulae show the specifications of static threads, and the initial memory and instruction sequence.

Inv ≡ True
Δ ≡ \{ main \sim (True, A₀, G₀), chld \sim (p₂ \land p₉, A₂, G₂) \}
Initial M ≡ \{ a \sim _0 b \sim _0 flag \sim _0 (flag + 1) \sim _0 \}
Initial I ≡ jd begin

Inv is simply set to True. The partial correctness of the gcd algorithm is inferred by the fact that p₂ is established at the entry point and G₂ is satisfied. The code heap specification is separated and given as preconditions (surrounded by \{-\}) at the beginning of each code block.

Two child threads are created using the same static thread chld. They use thread arguments to distinguish their task. Correspondingly, the assumption A₂ and G₂ of the static thread chld also use the thread argument to distinguish the specification of different dynamic copies. The child thread does not change its assumption and guarantee throughout its lifetime. Therefore we omit A₂ and G₂ in the code heap specifications. Since we insert yield instructions at every program point, every local guarantee is simply G₂, which is also omitted in the specifications.

CMAP allows the main thread to have different assump-
tions and guarantees. At the data initiation stage, the main thread assumes no threads in the environment changes a, b and the flags, and guarantees nothing, as shown in $A_0$ and $G_0$ (see Figure 9). After creating child threads, the main thread changes its assumption and guarantee to reflect the changing environment. The new assumption is just the disjunction of the previous assumption and the guarantee of the new thread (instantiated by the thread id and thread argument), similarly the new guarantee is the conjunction of the previous guarantee and the new thread’s assumption. Readers can refer to the FORK rule and check the validity.

This example also shows how thread join can be implemented in CMAP by synchronization. We use one flag for each thread to indicate if the thread is alive or not. The assumptions and guarantees of the main thread also take advantages of these flags so that it can weaken its guarantee and strengthen its assumption after the child threads die.

4.2 Unbounded Dynamic Thread Creation

In Figure 12 we show a small program which spawns child threads within a while loop. This kind of unbounded dynamic thread creation cannot be supported using the cobegin/coend structure. We show how such a program is specified and verified using our logic. To simplify the specification, we trivialize the functionality of each child thread, who just increases the corresponding array entry by 1.

We assume the high-level program works in a preemptive mode. Figure 13 shows the CMAP implementation, where yield instructions are inserted to simulate the preemption.

The program specifications are also shown in Figure 13. For each child thread, it is natural to assume that no other threads will touch its share of the data entry and guarantee that other threads’ data entry will not be changed. For the main thread, it assumes at the beginning that no other threads in the environment will change any of the data entries. It changes its assumption and guarantee while new child threads are created.

Specifying the loop body is not easy. At first glance, $A_1$ and $G_1$ can be defined the same as $A_3$ and $G_3$, respectively. However, this does not work because our FORK rule requires $G \Rightarrow G_1$, which cannot be satisfied. Instead, our $A_1$ and $G_1$ are polymorphic over the loop index $r_0$, which reflects the composition of the changing thread environments.

4.3 General Properties for Concurrent Programs

Yu and Shao [22] have shown by examples that CCAP is expressive enough to verify general properties of concurrent programs, like mutual exclusion and deadlock freedom. CMAP, as an extension of CCAP with dynamic thread creation and better modularity, is strictly more expressive than CCAP (it is trivial to translate the CCAP code and verification to PCC packages in CMAP). Therefore, CMAP also applies for those CCAP examples. In Appendix C, we give one more example which uses thread lock to achieve the mutual exclusion in the Producer-Consumer program.

5 Related Work and Conclusion

We have presented a certified programming framework for verifying multi-threaded assembly code with unbounded dynamic thread creation. Our work is related to two directions of research: concurrency verification and PCC. The rely-guarantee method [11, 16, 17, 1, 7, 22] is one of the best studied technique for compositional concurrent program verification. However, most of the work on it are based on high-level languages or calculus, and none of them support unbounded dynamic thread creation. On the other
hand, many PCC frameworks [13, 12, 2, 10, 21, 4, 22] have been proposed for machine/assembly code verification, but most of them only support sequential code. The only intersection point of these two directions is the work on CCAP [22], which applies the R-G method at the assembly level to verify general concurrency properties, like mutual exclusion and deadlock-freedom. Unfortunately, CCAP does not support dynamic thread creation either.

There are also much work using type-based approaches for concurrent programming [5, 6, 8, 9]. Different with logic-based approaches to verify general program properties, they use specialized type systems to reason about specific properties, including races, deadlocks, and atomicity.

CMAP extends previous work on R-G method and CCAP with dynamic thread creation. We unify the concepts of a thread’s assumption/guarantee and its environment’s guarantee/assumption, and allow a thread to change its assumption and guarantee to track the changing environment caused by dynamic thread creation. Code segments in CMAP can be specified and verified once and used in multiple threads with different assumptions and guarantees, therefore CMAP achieves better modularity than CCAP. Some practical issues, such as argument passing at thread creation, thread local data, and multiple invocation of one copy of thread code, are also discussed to support practical multi-threaded programming.

The thread primitives in CMAP are higher-level pseudo instructions. They can be replaced by system call to certified thread library, which is part of our ongoing work. Also we separate issues in multi-threaded programming from the embedded code pointer problem which is addressed in a companion paper [14]. Applying that framework to our thread model will be part of the future work.

References


A The Rest of CMAP Inference Rules

We first introduce the following shorthands:

\[ S(r) \equiv S.R(r) \]
\[ A S \equiv A (S.R) (S.M) \]
\[ G S \equiv G (S.R) (S.M) \]
\[ g S \equiv g (S.R) (S.M) \]

We establish the following (universal) rules.

\[ \Phi \vdash T \]
\[ (\forall k, A_k, G_k) = \theta_k = \Delta(h_k), \ I_k = \Theta(h_k), \ \forall h_k \in dom(T) \]
\[ \forall (M, R), (M', R') (M, R) \Rightarrow A_k \ R M' \Rightarrow p_k \ (M', R) \]
\[ \forall (\Delta, \Psi); (p_k, G_k, A_k, G_k) \vdash I_k \]

\[ \Phi \vdash \Box \]
\[ \text{dom}(\Psi) = \text{dom}(\Box) \]
\[ \text{dom}(\Delta); \Psi(\epsilon) \vdash \Box(\epsilon), \forall \epsilon \in \text{dom}(\Psi) \]

\[ \Phi; (p, g, A, G) \vdash \epsilon \]

\[ \epsilon \in \{ \text{add} \ r_d, r_s, r_t, \text{sub} \ r_d, r_s, r_t, \text{movi} \ r_d, w \} \]
\[ \forall (\Delta, \Psi); (p_k, G_k, A_k, G_k) \vdash \epsilon \]

\[ \epsilon = \text{id} \ r_d, r_s(w) \]
\[ \forall (\Delta, \Psi); (p_k, G_k, A_k, G_k) \vdash \epsilon \]

\[ \epsilon = \text{st} \ r_d(w), r_s \]
\[ \forall (\Delta, \Psi); (p_k, G_k, A_k, G_k) \vdash \epsilon \]

\[ (\text{ST}) \]

\[ (\text{LD}) \]

\[ (\text{SIMP}) \]

\[ (\text{CDHP}) \]

\[ (\text{THRDS}) \]

\[ (\text{BEQ}) \]

\[ (\text{BGT}) \]

\[ (\text{T}) \]

\[ (\text{I}) \]

B The Soundness of CMAP

The soundness of CMAP inference rules with respect to the operational semantics of the machine is established following the syntactic approach of proving type soundness [20]. From the “progress” and “preservation” theorems, we can guarantee that a well-formed program under compatible assumptions and guarantees, the current instruction sequence will be able to execute without getting “stuck.” Furthermore, any safety property derivable from the global invariant will hold throughout the execution.

Theorem B.1 (Progress)

\[ \Phi = (\Delta, \Psi). \text{ If there exists } \Theta, p, g, A, \text{ and } G \text{ such that } \Phi; \Theta; (p, g, A, G) \vdash (S, Q, T, C, I), \text{ then } (\text{Inv } S), \text{ and there exists a program } \bar{P} \text{ such that } (p, g, A, G) \rightarrow \bar{P}. \]

Proof sketch: By induction over the structure of \( \models \). (Inv S) holds by the assumption and the inversion of the rule PROG. In the cases where \( \models \) starts with add, sub, movi, fork, exit, or yield, the program can always make a step by the definition of the operational semantics (recall in our abstract machine we always assume there is an “idle” thread in \( Q \), therefore the instructions fork and exit can go through). In the cases where \( \models \) starts with \text{id} and \text{st}, the side conditions for make a step, as defined by the operational semantics, are established by the rules LD and ST. In the cases where \( \models \) starts with \text{bgt} or \text{beq}, or where \( \models \) is \text{id}, the operational semantics may fetch a code block from the code heap; such a code block exists by the inversion of the rule CDHP.

Theorem B.2 (Preservation)

If \( \Phi; \Theta; (p, g, A, G) \vdash P \) and \( P \rightarrow \bar{P} \), where \( P = (S, Q, T, C, I) \) and \( \bar{P} = (S, Q, T, C, I) \), then there exist \( \Theta, \bar{P}, g, A, \text{ and } G \text{ such that } \Phi; \Theta; (\bar{P}, g, A, G) \vdash P. \)

Proof sketch. By the assumption \( \Phi; \Theta; (p, g, A, G) \vdash P \) and the inversion of the rule PROG, we know that

1. \( (\text{Inv}, \Delta, \Psi) = \Phi, (M, R) = S, t = S(rt); \)
2. \( \Phi \vdash T; \)
3. \( \Phi \vdash C; \)
4. \((\text{Inv} \land p) \triangleright S;\) 
5. \(\Phi; (p, g, A, G) \vdash I;\) 
6. \(\Phi; (g, S) \vdash Q;\) 
7. \(\text{NI}(\Theta(t \sim (p, A, G)), Q(t \sim (R, I)))\) (see Section 3.4 for the definition of NI).

We prove the theorem by induction over the structure of \(I\). Here we only give the detailed proof of cases where \(I\) starts with fork and yield. Proof for the rest cases are trivial.

Case \(I = \text{fork} h, r; I'\).

By the operational semantics, we know that \(\tilde{S} = S, \tilde{Q} = \{t', \ldots, t_{15} \sim \ldots \sim rt \sim t', ra \sim S(x)\}, I' = T(h), t' \notin \text{dom}(Q)\) and \(t' \neq t\). According to 5 and the inversion of the fork rule, we know that there exist \(p', \tilde{A}', G', A''\) and \(G''\) such that

\[
\begin{align*}
\text{f.1} & : (p', \tilde{A}', G') = \Delta(h); \\
\text{f.2} & : A \Rightarrow A'', G'' \Rightarrow G, (A \lor G'') \Rightarrow \tilde{A}', \text{and } G'' \Rightarrow (G \land A''); \\
\text{f.3} & : \forall (M, R) . \forall \tilde{M}' . (\text{Inv} \land p) (M, R) \Rightarrow \tilde{R}(rt) \neq \tilde{R}'(rt) \Rightarrow \tilde{R}(x) = \tilde{R}'(ra) \Rightarrow g R M M' \Rightarrow p' (M', R'); \\
\text{f.4} & : (\text{Inv}, \Delta, \Psi); (p, g, A'', G'') \vdash I''.
\end{align*}
\]

Then we let \(\bar{\Theta} = \Theta(t \sim (p', \tilde{A}', G'))\), \(\bar{p} = p, \bar{g} = g, \bar{A} = A'', \text{and } \bar{G} = G''\).

According to the rule PROG, to prove \(\Phi; \Theta; (\bar{p}, \bar{g}, \bar{A}, \bar{G}) \vdash \bar{P}\), we need prove the following:

- \(\Phi \vdash T\) and \(\Phi \vdash C\). They trivially follow 2 and 3. Since these two conditions never change, we will omit the proof of them in the following cases.

- \((\text{Inv} \land p) \triangleright S;\) By 4.

- \(\Phi; (p, q, A', G') \vdash I''.\) By f.4.

- \(\Phi; (g, S) \vdash Q;\) By 6 and the inversion of the rule DTHRDS, we know that we only need check the following for the new thread \(t':\)

\[
\begin{align*}
& - \forall M', M''. (\text{Inv} \land p') (M', R') \Rightarrow A' R' M' M'' \Rightarrow p' (M'', R'). \text{ This follows 2 and the inversion of the rule THRDS.} \\
& - (\text{Inv}, \Delta, \Psi); (p', G', A'', G') \vdash I''. \text{ This follows 2 and the inversion of the rule THRDS.} \\
& - \forall M'' . g S M'' \Rightarrow p' (M'', R'). \text{ By 4, and f.3.} \\
& - \text{NI}(\bar{\Theta}(t \sim (p, A', G'))), \bar{Q}(t \sim (R, I''))\}. \text{ By 4, 7, and f.2.}
\end{align*}
\]

Case \(I = \text{yield}; I''\).

By the operational semantics, we know that \(\tilde{S} = (M, R'), \tilde{Q} = (Q \{t \sim (R, I'')\}) \setminus \{t\}, \text{ and } I' = I\), where \(t \in \text{dom}(Q)\) and \((R', I') = Q(t)\) or \(t = R(rt)\) and \((R', I') = (R, I'')\).

If \(t = R(rt)\), the yield instruction is like a no-op instruction. The proof is trivial. So we just consider the case that \(t \in \text{dom}(Q)\) and \((R', I') = Q(t)\).

We let \(\bar{\Theta} = (\Theta(R(rt) \sim (p, A, G))) \setminus \{t\}, \bar{p} = p', \bar{g} = g', \bar{A}' = A', \text{ and } \bar{G} = G', \text{ where } (p', A', G') = \Theta(t).\)

According to the rule PROG, to prove \(\Phi; \bar{\Theta}; (\bar{p}, \bar{g}, \bar{A}, \bar{G}) \vdash \bar{P}\), we need prove the following:

- \((\text{Inv} \land p') (M, R')\) (note that Inv is independent of R).

- By 6 and the inversion of the DTHRDS rule, we know that \(\forall M'', g S M'' \Rightarrow p' (M'', R')\), also by 4, and the inversion of the YIELD rule we get \(g S M\). Therefore we can prove \(p' (M, R')\).

- \(\Phi; (p', G', A', G') \vdash I'.\) By 6 and the inversion of the rule DTHRDS.

- \(\Phi; \bar{\Theta}; (G', (M, R')) \vdash \bar{Q}.\) To prove this, we only check the following for the yielding thread:

\[
\begin{align*}
& - \forall M'', (\text{Inv} \land p)(M', R) \Rightarrow A R M' M'' \Rightarrow p(M'', R). \text{ This follows 5 and the inversion of the rule YIELD;} \\
& - (\text{Inv}, \Delta, \Psi); (p, g, A, G) \vdash I''. \text{ This follows 5 and the inversion of the rule YIELD.} \\
& - \forall M'', G' R' M M'' \Rightarrow p (R, M''). \text{ By 7 we know } \forall M'', G' R' M M'' \Rightarrow A R M' M''. \text{ Therefore, we only need prove:} \\
& \forall M''. A R M M'' \Rightarrow p (M'', R). \text{ By 5, the inversion of the rule YIELD, and 4 we get it.}
\end{align*}
\]

- \(\text{NI}(\bar{\Theta}(t \sim (p', A', G'))), \bar{Q}(t \sim (R', I')))\). This trivially follows 7.

\[\square\]

C The Producer-Consumer Program

In Figure 14, we show the high-level pseudo code for the Producer-Consumer program. The producer and consumer share the buffer buff and the counter c of unconsumed objects in the buffer. Pointers head and tail t are used by the producer and consumer, respectively, to access the buffer. We use the variable \(l\) to implement thread lock for mutual exclusive access to buff and c.

Figure 15 shows the consumer code. The code for producer is omitted because of the symmetry. Since here we only interested in the protection of the critical region, we
Variables:
```
int[5] buff;
int h, t, c, l;
int vi, v2;
```

Initially:
```
h = 0 \land t = 0;
c = 0 \land l = 0;
```

Producer:
```
produce(v1);
while(true){
lock_acq(v1);
if (c < 5){
c := c + 1;
buff[h] := v1;
if (h < 4)
h := h + 1;
else
h := 0;
lock_rel();
produce(v1);
}else
lock_rel();
}
```

Consumer:
```
while(true){
lock_acq();
if (c > 0){
c := c - 1;
v2 := buff[t];
if (t < 4)
t := t + 1;
else
t := 0;
lock_rel();
comsume(v2);
}
```

Figure 14. The High Level Program

```
csnm := \{(True, A_c, G_c)\} 
jd acq
yield
movi r0, 0 
acq := \{(True, g)\}
yield
movi r1, c 
lid r2, r1(0) 
mov r0, 0 
bgt r2, r0, 0, deq 
mov r1, 1 
lid r2, r1(0) 
beq r0, r2, grab 
deq := \{(p4 \land p2 \land p5)\} 
yield 
jd acq 
movi r0, 1 
grab := \{(p4, g1)\}
s1 r2(0), rt 
yield 
jd cont 
movi r1, buf 
mov r2, t 
rel := \{(p4 \land p1 \land p2 \land G_p)\}
mov r0, 0 
lid r3, r2(0) 
add r4, x1, r3 
sl r0(1), r0 
lid r5, x4(0) 
jd acq
movi r6, v2 
sl r6(0), r5 
inc := \{(p4, g)\}
yield 
movi r1, 4 
mov r0, 1 
bgt r1, r3, inc 
add r4, r3, r0 
sl r2(0), r4 
yield 
jd rel
```

Figure 15. Spec. and The Consumer Code

also omit the part of the code that consumes the value got from the buffer: we assume it is simply discarded by the consumer.

The program specifications are shown in Figure 16. The assumption and guarantee of the consumer is given by $A_c$ and $G_c$ ($A_p$ and $G_p$ are for the producer). They also reflect the semantics of thread “lock”: if the consumer holds the lock at the time it yields control, it assumes that the shared resource will not be changed by others; also it guarantees not to change the shared resource if the lock is held by other threads.

The resource invariant $RI$ describes the property of the shared resource maintained along the execution of the program, which is also used as the global invariant $Inv$. In particular, it enforces the property that the shared resources are in good shape outside of the critical region.

```
p_1 \equiv (0 \leq t < 5) \land (0 \leq h < 5) \land (0 \leq c < 5)
p_2 \equiv (t = h) \land (c = 0 \lor c = 5) \lor (t > h) \land (t - h + c = 5)
\land (t < h) \land (h - t = c)
RI \equiv p_1 \land (1 = 0 \Rightarrow p_2) \land RI' \equiv RI'(v' : v'/v)
Inv \equiv RI'
```

Figure 16. Program Specifications