Modular Specification and Verification of Interprocess Communication

Eyad Alkassar*, Ernie Cohen†, Mark Hillebrand†, and Hristo Pentchev*
*Saarland University, Saarbrücken, Germany
{eyad,pentchev}@wjpserver.cs.uni-saarland.de
†European Microsoft Innovation Center (EMIC GmbH), Aachen, Germany
{ecohen,mahilleb}@microsoft.com

***Abstract*—The usual goal in implementing IPC is to make a cross-thread procedure call look like a local procedure call. However, formal specifications of IPC typically talk only about data transfer, forcing IPC clients to use additional global invariants to recover the sequential function call semantics. We propose a more powerful specification in which IPC clients exchange knowledge and permissions in addition to data. The resulting specification is polymorphic in the specification of the service provided, yet allows a client to use IPC without additional global invariants. We verify our approach using VCC, an automatic verifier for (suitably annotated) concurrent C code, and demonstrate its expressiveness by applying it to the verification of a multiprocessor flush algorithm.**

I. Introduction

Procedural abstraction—the ability for the caller of a procedure to abstract a procedure call to a relation between its pre- and poststates—is one of the most important structuring mechanisms in all of programming methodology. The central role of procedural abstraction is reflected in the fact that it is built into not only all modern imperative languages, but also into most program logics and verifiers for such languages. However, in a concurrent or distributed system, procedure calls between threads are provided only indirectly through system calls or libraries for interprocess communication (IPC). This begs the question of how such libraries might be specified so as to provide procedural abstraction to their clients, and how such libraries can be verified to meet these specifications. In this paper, we consider the problem in the context of multithreaded C software, with threads executing in a single shared address space.

To see why this problem is nontrivial, consider a simple implementation where all data is passed through shared memory, and where each ordered pair of caller-callee threads share a mailbox at a fixed address. The caller makes a call by creating a suitable call record in memory (including identification of which procedure to execute, values of the call parameters, and a place to put the return value), writes the address of this record into the mailbox going to the callee, and calls an IPC function to signal the callee. The callee, on receiving the signal, reads the address of the call record from the mailbox, reads the memory to get the call parameters, executes the call, and signals the caller. Note that all memory accesses are sequential; the only synchronization necessary is provided by the IPC layer.

Now, it's not hard to see that the IPC layer is providing functionality similar to a split binary semaphore, with the call records playing the role of the lock-protected data, and the data invariant given by the semantics of the various procedure calls. Thus, a specification for semaphores would provide a natural starting point for a specification for IPC. However, in classical program verification, semaphore operations are specified by their effect on global ghost state; making use of such a specification requires additional global invariants to capture how the clients use each semaphore. Using this kind of specification for IPC would force the client of the remote procedure call to use these global invariants on both call and return. This fails to faithfully capture the local character of procedural abstraction.

A second possibility is to encapsulate these global invariants inside the IPC layer. For example, the IPC specification could be strengthened to include the pre- and post-conditions of the procedure call. This is the sort of specification one would find in a local logic, such as concurrent separation logic (CSL). But such logics typically cannot specify generic semaphores, because the semaphore code has to be polymorphic in both the encapsulated data and the data invariant.[1] Similarly, taking this approach with the IPC code requires the specification of the code to be polymorphic in the specification of the material being passed between caller and callee.

We propose a different approach to specifying and verifying IPC that allows the recovery of procedural abstraction. The key idea is that IPC routines transfer ghost objects that own the call records, and whose invariants capture the pre- and post-conditions of the procedures. (This is possible because we allow object invariants to mention arbitrary parts of the state, with a semantic consistency check that guarantees the stability of each object invariant while the object exists.) The "contract" between caller and callee is expressed in ghost data as a binary relation between call objects and return objects. The IPC routines can transfer ownership of the ghost objects without knowing their types, making the transport suitably polymorphic. This ghost scaffolding, combined with the (fixed) specification of the IPC routines, yields for the client the sequential procedural abstraction provided by the

[1] A recent proposal [7] extends CSL with a facility similar to VCC ghost objects, which should allow to do constructions similar to the one in this paper.

application function.

We have used this approach to specify and verify an IPC layer, and illustrate its application to a multiprocessor flush algorithm. The implementation was derived from a real verification target, the inter-processor interrupt (IPI) routines of Microsoft's Hyper-V$^{\text{TM}}$ hypervisor. All specification and proofs given here have been carried out using VCC, an automatic verifier for (suitably annotated) concurrent C.[2] VCC provides the first-class ghost objects needed to carry out our approach, while allowing the approach to be applied to real implementation code.

A. Related Work

The correctness of IPC has been tackled in the context of microkernel verification. For example, the IPC implementations of the seL4 [9] and VAMOS [6] kernels have been formally verified against their respective ABIs. These projects focused on implementation correctness rather than client usability, and specify solely data transfer.

The application of VAMOS IPC provided in [1] shows the shortcomings of this approach: there, correctness statements of the remote procedure calling (RPC) library argue simultaneously on the sender/receiver pair instead of using thread-local reasoning.

A number of formalisms were applied to specification and verification of interprocess communication in the context of the RPC-Memory Specification Case Study [3]. None of the submitted solutions attempted to provide general-purpose sequential procedural abstraction.

In [8] a verification framework for threads and interrupt handlers based on CSL is described. This work is similar to ours, as both the implementation of (thread-switching) primitives and clients using them, are verified. When threads switch, ownership is transfered and some global invariant on shared data is checked. In contrast to our work the client code is interactively verified in two different logics, whereas in our approach both are verified seamlessly and automatically in the same proof context.

B. Overview

The paper is structured as follows. In Section II we outline main VCC concepts. In Section III we present an IPC algorithm with polymorphic specification, which we use in Section IV to implement and verify a TLB flush protocol. In Section V we extend these results to multiple senders and receivers as required for the implementation of interprocessor interrupt (IPI) protocols used in real, multiprocessor hypervisors. In Section VI we conclude.

II. VCC Overview

In this section, we give a brief overview of VCC. More detailed information and references can be found through the VCC homepage [10]. To understand the VCC view of the world, it is helpful to think of verification in a pure object model, which is used to interpret the C memory state. Thus, we first describe VCC concepts in terms of objects, and then describe how this is applied to C.

Table I shows a syntax overview of the constructs required for our IPC design presented in the following sections.

A. Objects

In VCC, the state is partitioned into a collection of objects, each with a number of fields. Objects have addresses, so fields can be (typed) object references. Each object has a 2-state invariant, which is expected to hold over any state transition. These invariants can mention arbitrary parts of the state. However, when checking an atomic update to the state, instead of checking the invariants of *all* objects we want to check the invariants of only the updated objects. We justify this by checking, for each object type, that starting from a state in which all object invariants hold, a transition that breaks the invariant of an object of that type must break the invariant of some modified object (not necessarily of that type); such invariants are said to be *admissible*. (In addition, we have to check that stuttering from the poststate of a transition preserves all invariants of all objects.) Both requirements are checked for each object type when the type is defined; this check makes use of type definitions, but not of program code. Details can be found in [5].

Within an object invariant, the (2-state) invariant of other objects can be referred to.[3] A commonly used form of this is *approval*: we say that an object o *approves* changes to another object's field $p{\to}f$, if p has a 2-state invariant stating that $p{\to}f$ stays unchanged or the invariant of o holds. In other words, any change to $p{\to}f$ requires checking the invariant of o. Approval is used to express object dependencies or build object hierarchies, e.g., VCC's ownership model.

Since it is unrealistic to expect objects to satisfy interesting invariants always (e.g., before initialization or during destruction), we add to each object a Boolean ghost field ***closed*** indicating whether the object is in a "valid" state. Implicitly, the 2-state invariants declared with an object type are meant to hold only across state transitions in which the object is closed in the prestate and/or the poststate. Each object field is classified as either sequential or volatile. Volatile fields can change while the object is closed, while sequential fields cannot. (That is, for each sequential field, there is an implicit object invariant that says that the field does not change while the object is closed.)

Each object has an *owner*, which is itself an object. It is a global system invariant that open objects are owned only by threads, which are regular objects. In the context of a thread t, a closed object owned by t is said to be *wrapped*, while an open object owned by t is said to be *mutable*. Threads themselves have invariants; essentially, the invariant of a thread t says that any transition that does not change the state of t leaves unchanged (i) the set of objects owned by t, (ii) the fields of its mutable objects, (iii) the sequential fields of its wrapped

[2]Sources are available at http://www.verisoftxt.de/PublicationPage.html.

[3]This implicitly makes object invariants recursive; to guarantee that all object invariants have a consistent interpretation, we allow such references to occur only with positive polarity.

VCC Keyword	Description
Basics	
this	self-reference to object (used in type invariants)
invariant*(p)*	type invariant with property *p*
old*(e)*	evaluates *e* in prestate (of function, loop, or 2-state invariant)
closed*(o)*	object *o* closed; invariants of *o* guaranteed to hold
inv*(o)*	evaluates to (2-state) invariant of *o*
approves*(o, $f_1, \ldots, f_n$)*	changes of fields $f_1,...,f_n$ require check of *o*'s invariant: $(\bigvee_i$ ***old****(f_i)*$\neq f_i) \implies$ ***inv****(o)*
atomic*($o_1, \ldots, o_n$)*$\{s;\}$	marks atomic execution of *s*; updates only volatile fields of $o_1, \ldots, o_n$
ref_cnt*(o)*	number of claims that depend on *o*
claims*(c,p)*	invariant of claim *c* implies *p*
spec*(...)*	wraps ghost code and parameters
$\forall$*(T t;...)*	universal quantification
$\exists$*(T t;...)*	existential quantification

(a) Basic Keywords

VCC Keyword	Description
Ownership	
owner*(o)*	owner of object *o*
owns*(o)*	set of objects owned by object *o*
wrapped*(o)*	*o* closed and owned by current thread
mutable*(o)*	*o* not closed and owned by current thread
set_closed_owner*(o,o')*	sets owner of *o* to *o'* and extends ownership of *o'* by *o*
giveup_closed_owner*(o,o')*	make *o* wrapped and remove it from the ownership of *o'*
Function Contracts	
requires*(p)*	precondition
ensures*(p)*	postcondition
writes*($o_1, \ldots, o_n$)*	function writes to objects o_i
Spec Types	
mathint	mathematical integers
claim_t	claim pointers
T2 map[T1]	map from *T1* to *T2*
λ*(T1 t1;...)*	lambda expression over *t1*

(b) Ownership, Function Contracts, Spec Types

TABLE I: VCC Keywords

objects, and (iv) the (volatile) fields of closed objects approved by *t* (we call such fields *thread-approved*). Each object *o* implicitly contains an invariant that says that its owner (as well as its owner in the prestate) approves any change to the field *o*→***closed*** and to the set of objects owned by *o*.[4]

The *sequential domain* of a closed object is the smallest set of object fields that includes the sequential fields of the object and, if its set of owned objects is declared as nonvolatile, the elements of the sequential domains of the objects that it owns. Intuitively, the values of fields in the sequential domain of *o* are guaranteed not to change as long as *o* remains closed.

Within program code, each memory access is classified as ordinary or atomic. An ordinary write is allowed only to fields of mutable objects; an ordinary read is allowed only to fields of mutable objects, to nonvolatile fields of in the sequential domain of a wrapped object, and to volatile fields of objects that are closed if changes to the field are approved by the reading thread. In an atomic operation, all of the objects accessed have to be known to be mutable or closed (i.e., not open and owned by some other thread), only volatile fields of closed objects may be written, and the update must be shown to preserve the invariants of all updated objects. Before each atomic operation, VCC simulates running other threads by forgetting everything it knows about the state outside of its sequential domain; standard reduction techniques [4] can be used to show that we can soundly ignore scheduler boundaries at other locations.

B. Ghost Objects

VCC verifications make heavy use of *ghost* data and code (surrounded by ***spec****()*), used for reasoning about the program but omitted from concrete implementation. VCC provides ghost objects, ghost fields of structured data types, local ghost variables, ghost function parameters, and ghost code. C data types are limited to those that can be implemented with bit strings of fixed length, but ghost data can use additional mathematical data types, e.g., mathematical integers (***mathint***) and maps. VCC checks that information cannot flow from ghost data or code to non-ghost state, and that all ghost code terminates; these checks guarantee that program execution including ghost code simulates the program with the ghost data and ghost code removed.

C. Claims

A ghost object can be used as a first-class chunk of *knowledge* about the state, because the invariant of the object is guaranteed to hold as long as the object is closed. In particular, the owner of the object does not have to worry about the object being opened by the actions of others, so it can make use of the object invariant whenever it needs it. Being a first-class object, the chunk can be stored in data structures, passed in and out of functions, transfered from thread to thread, etc. Because they are so useful, VCC provides syntactic support for these chunks of knowledge, in the form of *claims*. Claims are similar to counting read permissions in separation logic [2], but are first-class objects; this allows claims to approve changes, be claimed, or even claim things about themselves.

Typically, a claim depends on certain other objects being closed; it is said to "claim" these objects. Since objects are usually designed to be opened up eventually, these "claimed" objects must be prevented from opening up as long as the claim is closed. Concretely, this can be implemented in various ways, the most obvious being for the dependee to track the count ***ref_cnt****(o)* of claims that claim *o*, and allowing *o* to be opened only when ***ref_cnt****(o)* is zero, cf. [5]. In constructing a claim, the user provides the set of claimed objects and invariant of the claim; VCC checks that this invariant holds and is

[4]By default, the set of objects owned by *o* is nonvolatile, and so cannot change while *o* is closed. This can be overridden by declaring ***vcc****(volatile_owns)* in the type definition of *o*.

preserved by transitions under the assumption that the claimed objects are closed (this check corresponds to the admissibility check if the claim was declared with an explicit type). Any predicate implied by this invariant is said to be "claimed" by the claim; this allows a client needing a claim guaranteeing a particular fact to use any claim that claims this fact (without having to know the type of the claim); to make this convenient, VCC gives all claims the same type (***claim_t***); we can think of an additional "subtype" field as indicating the precise invariant.

D. Function Contracts and Framing

Verification in VCC is *function-modular*; when reasoning about a function call, VCC uses the specification of the function, rather than the function body. A function specification consists of preconditions (of the form ***requires****(p)*), postconditions (of the form ***ensures****(p)*, where *p* is a 2-state predicate, the prestate referring to the state on function entry), and writes clauses (of the form ***writes****(o)*, where *o* is an object reference or a set of object references). VCC generates appropriate verification conditions to make sure that the writes clauses are not violated.

E. Binding to C

The discussion above assumed that we are in a world of unaliased objects. To deal with the real C memory state, VCC maintains in ghost state a global variable called the *typestate* that keeps track of where the "real" objects are; these objects correspond to instances of C aggregate types (structs and unions). (Variables of primitive types that are not fields of such objects are put into artificial ghost objects or ghost arrays.) There are system invariants that (i) each memory cell is part of exactly one object in the typestate, (ii) if a struct is in the typestate, then each of its subobjects (e.g., fields of aggregate type) are in the typestate, and (iii) if a union is in the typestate, then exactly one of its subobjects is in the typestate. These invariants guarantee that if two objects overlap, then they are either identical or one object is a descendant of the other in the object hierarchy. When an object reference is used (other than as the target of an assignment), it is asserted that reference points to an object in the typestate. Thus, the typestate gets rid of all of the "uninteresting" aliasing (like objects of the same type partially overlapping).

III. A Polymorphic Specification of IPC

In this section we verify the implementation of a simple communication algorithm between two threads. The threads exchange data over a shared but sequentially accessed message box to which they synchronize access with a Boolean volatile notification flag. To verify the implementation's memory safety, an ownership discipline must be realized in which the ownership of the message box is transferred back and forth between the two threads. We extend this pattern by passing claims between the two threads, which we store in the message box. The properties of these claims can be configured by the clients, thus providing the desired polymorphic procedure call semantics for IPC.

```
spec(typedef struct vcc(record) InOut {
  unsigned val; mathint gval; claim_t cl; } InOut;)

typedef struct MsgBox {
  unsigned in, out;
  spec(InOut input, output;)
  invariant(input.val≡ in ∧ output.val≡ out)
  invariant(input.cl≠ output.cl ∧
    input.cl ∈ owns(this) ∧ ref_cnt(input.cl)≡ 0 ∧
    output.cl ∈ owns(this) ∧ ref_cnt(output.cl)≡ 0) } MsgBox;
```

Listing 1: Message Box Type with Invariants

There are various ways to structure annotations and, in particular, the definitions of ghost objects and invariants. At their core, all of these share information via volatile fields, pass on knowledge via claims or object invariants, and make use of thread-approved state for the two communication partners. We chose here a way that is easy to present but also extends cleanly to multiple senders and receivers (cf. Section V).

A. Scenario

We consider the scenario of two threads (0 and 1) exchanging data over a shared message box (of type *MsgBox*). The message box contains two fields (*in* and *out*) which are used for sending a request to the other thread and receiving back a response, respectively. The fields of the message box are nonvolatile and accessed sequentially. The message box is contained in another structure (of type *Mgr*), which additionally holds a volatile Boolean notification flag *n* used to synchronize access to the message box. Given the canonical conversion of Booleans to integers (where *0* and *1* are mapped to ***false*** and ***true***, respectively), this flag identifies the currently acting thread. If set, thread 1 is acting, i.e., preparing a response for thread 0 and posting a new request, and thread 0 may not access the message box. Otherwise, thread 0 is acting and thread 1 may not access the message box. Thread 0 may not clear the flag, and thread 1 may not set it.

The implementation has two functions. Both take a *Mgr* pointer and a thread identifier *a*. The function *snd()* is meant to be called by thread *a* when the notification flag equals *a*. It negates the notification flag, thus sending the response and a new request contained in the message box at that time to the other thread. The function *rcv()* waits in a busy loop until the notification flag equals *a* again, thus receiving the other thread's response (to a preceding *snd()* call) and a new request.

B. Message Box

Listing 1 shows the annotated definition of the message box type. As outlined above, we want to generalize information exchange to beyond the mere transferral of data (the fields *in* and *out* in the message box). We therefore define an abstract I/O type (*InOut*) that carries a ghost value *gval* of unbounded integer type, and a claim pointer *cl* in addition to the implementation data value *val* being transmitted.

An abstract input and output each are an invariant stored in ghost fields of the message box. We maintain an invariant that the input's and output's *val* fields match their implementation

```
spec(typedef struct vcc(volatile_owns) Actor {
  struct Mgr *mgr;
  volatile bool w;
  volatile InOut l_input, r_input;
  invariant(closed(this) ∨ ¬closed(mgr))
  invariant(approves(owner(this), w, l_input, r_input))
  invariant(approves(mgr, owns(this), w, l_input, r_input)) } Actor;)
```

Listing 2: Actor Type and Invariants

counterpart. We also require that the claims pointed to by the input's and output's *cl* fields do not alias and are owned by the message box with a zero reference count.

The latter fact is particularly important. Whoever owns the message box also controls the contained claims, and may make use of the knowledge / property they hold or destroy them. The main functionality of the verified algorithm is thus the transferal of ownership of the message box between the two threads, making sure that the contained data has the desired properties, as instantiated by the client.

C. Actors

The *Actor* type keeps track of the protocol state of a protocol participant. Listing 2 shows the annotated definition of this type. The actor has a nonvolatile pointer *mgr* to the manager, which will hold all protocol invariants. For admissibility reasons, the actor must promise to stay closed longer than *mgr*. All others fields are volatile and may be atomically updated while the actor remains closed. Such updates, however, must be approved by two parties: the manager *mgr*, which checks all the protocol invariants, and the owner of the actor, which is one of the communicating threads and exclusive writer of the fields. The actor is also used as an intermediate owner of the message box during ownership transferral. For this purpose, its owns set is also declared volatile as well as approved by *mgr* but *not* thread-approved, to enable foreign updates by other threads.

The three regular fields of the actor are used as follows. The wait flag *w* is active when the thread owning the actor is waiting for a response from the other thread. The fields *l_input* and *r_input* buffer (abstract) local and remote inputs, i.e., input to the last request sent to or received from the other thread (or, in other words: the evaluation of the call parameters from the caller's and callee's perspective, respectively). In contrast to the *input* fields of the message box itself, which may be opened and updated sequentially by the owning thread, these buffers can be admissibly referred to all the time and used in the protocol invariants.

D. Manager

Listing 3 shows the annotated declaration of the *Mgr* type. In addition to the implementation fields, we also add some ghost components for the verification: the maps *InP* and *OutP* encoding pre- and postconditions for the message exchange, and a two-element array *A* of actors.

The predicates are declared nonvolatile, which allows clients to deduce that they remain unchanged as long as the manager

```
typedef struct Mgr {
  volatile bool n;
  MsgBox msgBox;
  spec(Actor A[2];
    bool InP[bool][InOut];
    bool OutP[bool][InOut][InOut];)
  invariant(∀(unsigned a; a < 2 ⟹ closed(&A[a]) ∧ A[a].mgr≡ this))
  invariant(A[¬n].w)
  invariant(A[n].w
    ? &msgBox ∈ owns(&A[n]) ∧ OutP[n][A[n].l_input][msgBox.output]
      ∧
      A[¬n].l_input≡ msgBox.input ∧ InP[n][msgBox.input]
    : A[n].r_input≡ A[¬n].l_input) } Mgr;
```

Listing 3: Manager Type and Invariants

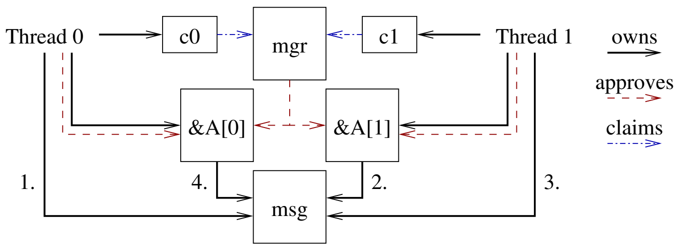

Fig. 1: Object Structure and Ownership Transfer

object is closed. They take a Boolean parameter identifying the actor and one resp. two abstract input-output values. The intention is that *InP[a][i]* is true iff i is a valid request for thread a (i.e., if the request meets the precondition of the service), and *OutP[a][i][o]* is true iff o is a valid response to a (valid) request i made by thread a (i.e., if the response meets the postcondition corresponding to the call). The IPI transport code is polymorphic with respect to these predicates, the concrete definition of which can be provided by the client at initialization.

We now describe the manager's invariants. As described above, each protocol partner a owns its corresponding actor *&A[a]*. The first invariant states that both actors remain closed and point back to the manager, which (in combination with the actor's approval invariants) allows us to admissibly talk about the actors in invariants here.

The remaining invariants define the protocol behavior. For an overview, refer to Fig. 1 depicting object structure and a protocol run starting from thread 0. In addition to the objects already introduced, each (client) thread i owns a claim *ci* that guarantees the manager structure to be closed. In phase 1, thread 0 owns the message box and may prepare its response and new request. In phase 2, ownership of the message box has passed from thread 0 to the actor of thread 1, waiting to be processed. Phases 3 and 4 are symmetrical: in phase 3 thread 1 prepares its response, which is then waiting to be processed in phase 4.

In addition to ownership, the protocol invariants restrict values for the actor fields. The second invariant states the non-acting thread, identified by the negated notification flag, must be waiting, i.e., have the wait flag of its actor set.

The third invariant refers to the acting thread, given by the

```
void snd(struct Mgr *mgr, bool a spec(claim_t c))
  requires(wrapped(c))
  requires(claims(c,closed(mgr)))
  requires(wrapped(&mgr→A[a]))
  ensures(wrapped(&mgr→A[a]))
  requires(¬mgr→A[a].w)
  requires(wrapped(&mgr→msgBox))
  requires(mgr→OutP[¬a][mgr→A[a].r_input][mgr→msgBox.output])
  requires(mgr→InP[¬a][mgr→msgBox.input])
  writes(&mgr→msgBox,&mgr→A[a])
  ensures(mgr→A[a].l_input≡ old(mgr→msgBox.input))
  ensures(mgr→A[a].w)
{
  atomic (c, mgr, &mgr→A[0], &mgr→A[1]) {
    assert(¬mgr→A[a].w ∧ mgr→A[¬a].w ∧ mgr→n≡ a);
    mgr→n = ¬a;
    spec(mgr→A[a].l_input = mgr→msgBox.input;
      mgr→A[a].w = true;
      set_closed_owner(&mgr→msgBox, &mgr→A[¬a]);
      bump_vv(&mgr→A[a]); /* technicality */
    )
  }
}
```

Listing 4: Send function with contract

notification flag. The fact that the acting thread is waiting indicates that the message box is still waiting to be processed by the acting thread. It holds a response to the acting thread's last request in the *output* field and a new request in the *input* field. In the corresponding invariant we state that (i) the message box is owned by the current actor, (ii) its output is valid with respect to the acting thread's last / locally-stored request, and (iii) the new input equals the local input buffer of the other thread and is valid for the acting thread. If the acting thread is not waiting, we require local and remote input buffers of the current and non-current actors, respectively, to match. Note that these input buffers are approved by the acting and non-acting threads, respectively. Thus, this condition states that request inputs may not be changed while the request has not yet been processed.

E. Operations

The (annotated) implementation and the contracts for the send and receive function are given in Listings 4 and 5. Both functions take a manager pointer *mgr*, an actor identifier *a*, and a claim *c* supplied as a ghost parameter stating that the manager is closed. They maintain that the identified actor is wrapped. To send to the other thread, the current thread's actor must be flagged as non-waiting, the message box must be wrapped and hold valid outputs and inputs to the other thread, just as we have seen in the manager invariant for the acting thread. Afterwards, the message box is unknown to be wrapped (the ***writes*** clause on *&mgr→msgBox* destroys that knowledge), but the input sent to the other thread is buffered in the local input field of the actor (and the current thread's actor is flagged as waiting).

Given a waiting actor, the receive function is guaranteed to return a wrapped message box, that contains a valid response for the old local request and a new valid request.

As a verification example consider the *snd()* function from

```
void rcv(struct Mgr *mgr, bool a spec(claim_t c))
  requires(wrapped(c))
  requires(claims(c,closed(mgr)))
  requires(wrapped(&mgr→A[a]))
  ensures(wrapped(&mgr→A[a]))
  requires(mgr→A[a].w)
  writes(&mgr→A[a])
  ensures(¬mgr→A[a].w)
  ensures(wrapped(&mgr→msgBox))
  ensures(mgr→OutP[a][old(mgr→A[a].l_input))][mgr→msgBox.output])
  ensures(mgr→A[a].r_input≡ mgr→msgBox.input)
  ensures(mgr→InP[a][mgr→msgBox.input])
{
  unsigned tmp;
  do
    invariant(mgr→A[a].w)
    invariant(wrapped(&mgr→A[a]))
    invariant(mgr→A[a].l_input≡ old(mgr→A[a].l_input))
    atomic (c, mgr, &mgr→A[0], &mgr→A[1]) {
      tmp = mgr→n;
      spec(if (tmp≡ a) {
        mgr→A[a].r_input = mgr→A[¬a].l_input;
        mgr→A[a].w = false;
        giveup_closed_owner(&mgr→msgBox, &mgr→A[a]);
        bump_vv(&mgr→A[a]); /* technicality */
      })
    }
  while (tmp≠ a);
}
```

Listing 5: Receive function with contract

Listing 4. VCC automatically verifies that its implementation fulfills the contract. The code consists of a single atomic update on the actors and the manager (where the closedness of the manager and the foreign actor is guaranteed by the claim *c*). The precondition on the wait flag, its thread-approval, and the manager's invariant allow to derive that the current thread is still not waiting, the other thread is waiting, and the notification flag equals *a* just before the atomic operation.[5] Also, the message box, which is in the sequential domain of the thread, must still be wrapped and continues to satisfy the communication preconditions. The notification flag is then flipped (changing the 'acting' thread) and the ghost updates ensure that the atomic update satisfies the manager's invariant (e.g., by transferring ownership of the message box from the current thread to the other thread's actor).

The verification of the *rcv()* function is similar. In addition to the atomic statement, appropriate invariants have to specified for the loop that polls on the notification flag.

IV. TLB Flush Example

We implement and verify a protocol for flushing translation look-aside buffers (TLBs) based on the communication algorithm from the previous section, demonstrating the expressiveness of its polymorphic specification.

TLBs are per-processor hardware caches for translations from virtual to physical addresses. These translations are defined by page tables stored in memory, which are asynchronously and non-atomically gathered by the TLBs (requiring multiple reads and writes to traverse the page tables). Since

[5]The assertion is for illustration only; VCC deduces it automatically.

```
spec(typedef struct Tlb {
    volatile mathint invalid, current;
    invariant(invalid ≤ current ∧
      old(invalid) ≤ invalid ∧ old(current) ≤ current)
} Tlb;)
```

Listing 6: TLB Model

```
typedef struct FlushMgr {
  struct Mgr mgr;
  spec(struct Tlb tlb;)
  invariant(&tlb ∈ owns(&mgr)))
  invariant(mgr.InP≡ λ(bool a; InOut i;
    a  ⟹  claims(i.cl, i.gval ≤ tlb.current)))
  invariant(mgr.OutP≡ λ(bool a; InOut i, o;
    a ∨ claims(o.cl, i.gval ≤ tlb.invalid)))
} FlushMgr;
```

Listing 7: Flush Manager Type and Annotations

translations are not automatically flushed in response to edits to page tables, operating systems must implement procedures to initiate such flushes on their own.

We think of page-table reads being marked with unique (increasing) identifiers and model each TLB as an object with two volatile counters,[6] cf. Listing 6. The *current* counter increases as the TLB gathers new translations. The *invalid* counter is a watermark for invalidated translations and is bumped (i.e., copied from the *current* field) when the associated processor issues a TLB flush.

Consider the scenario of two threads, the *caller* (thread 0) requesting the flush and the *callee* (thread 1) performing the flush. We implement this as follows: the caller sends a flush request by invoking the send primitive and subsequently polls for the answer by calling the receive primitive. On callee side, the thread polls via receive for new flush requests. When a flush request has been received, the callee issues a TLB flush operation, and signals back that the flush has been performed using the send primitive. After a completed flush operation, the flush client (e.g., the memory manager) wants to derive that the callee TLB's current *invalid* counter is larger or equal than the callee's *current* counter at the time of the flush operations.

We realize this scenario by embedding the IPC manager (and callee's TLB) into a *flush manager*, as shown in Listing 7. Apart from ownership, the invariants give meaning to the input and output predicates of the communication manager. The ghost value *i.gval* transmitted from the caller to the callee encodes which translations are meant to be flushed. For the callee (*a*≡ ***true***), the input predicate states that this value is less or equal than the *current* field of its TLB (since the callee could not possibly flush translation 'from the future', i.e., such a request could not be handled by the TLB flush semantics). For the caller (*a*≡ ***false***), the output predicate then states that the *invalid* field of the callee's TLB is greater or equal than the value, i.e., the requested flush has been performed. For the other cases the input and output predicates are trivially true.

[6]While this model is sufficiently detailed to express the semantics of (full) TLB flushes, extensions are needed for applications that go beyond that.

Based on this definition, the correctness of the functions *sendFlush()* and *receiveFlush()* at caller and callee side, respectively, can be proven. The main postcondition that is established by *sendFlush()* for the flush manager *fmgr* then is ***old****(fmgr→tlb.current)≤ fmgr→tlb.invalid.*

V. Interprocessor Interrupts

Interprocessor interrupts (IPI) are used in multicore operating systems or hypervisors to implement different synchronization and communication protocols. Via IPIs a thread executing on one processor can trigger the execution of interrupt handlers (here: NMI handlers) on other processors. Using IPIs, a communication protocol can be implemented, in which a caller thread sends work requests to other processors, the callees. Such an IPI protocol is part of the Verisoft XT academic hypervisor, where it may be used for different work types, e.g., for TLB flushing. Thus a polymorphic specification is desirable.

By expanding the simple communication pattern introduced previously, we specified and verified the IPI protocol (and on top of it a TLB flushing protocol) for the academic hypervisor. There are several differences between the previous version of the algorithm and the IPI protocol:

- **More communication partners.** In the simple case we had a single sender and a single receiver. Now we have multiple communication partners, where one sender may invoke an IPC call on many receivers, and where each receiver may be invoked by many senders at the same time.
- **No receiver polling.** The callees in the IPI scenario do not poll for messages. Rather the caller invokes the callee by triggering an IPI. This is done by writing registers of the advanced programmable interrupt controller (APIC), which delivers the interrupts to other processors. In the work at hand we do not yet model this hardware device.
- **More concurrency.** In the new setting we have another source of concurrency, NMI handlers which may interrupt the execution of ordinary threads. Basically, the NMI handler code always acts as receiver or callee and the thread code as sender or caller.
- **Interlocked hardware operations.** Interlocked bit operations are required to atomically access bit vectors which may be written and read concurrently by many threads/ handlers.

A. Implementation

Since multiple senders can send requests to multiple receivers, we need a notification bit for each sender/receiver pair. This is implemented by introducing one notification mask per processor. Each bit of such a mask is associated with a specific sending processor. Thus, a sender signals a request by setting its bit in the receiver's notification mask. When finishing the work, the receiver clears that bit. Many senders and receivers can write the same notification mask in parallel, requiring the use of interlocked bit operations.

Similarly, we need one mailbox for each sender/receiver pair. Note, that for each processor pair we need two mailboxes, since both may send messages to each other simultaneously.

In the sending code a while-loop iterates over the set of intended receivers (encoded in a bit mask). In each iteration, first the mailbox is prepared, and then by using an interlocked OR-operation, atomically, the corresponding bit in the receiver mask is set to *1* and the mask is compared with *0*. If this check evaluates to true, an IPI for the receiving processor is triggered via the APIC. Otherwise, nothing has to be done, since some other sender already triggered the interrupt, and the handler has not returned yet.

In the receiving code (implemented as an NMI handler) a while-loop iterates on (possibly multiple) sender requests as long as the receiver's notification mask is not *0*. Once the work for one sender is done, the corresponding bit in the notification mask is cleared by an interlocked AND-operation.

B. Specification

The specification pattern of Section III can be straightforwardly applied to the IPI protocol. The number of ghost objects scales linearly with the number of processors. The structure and the invariants of message boxes (with their ghost fields encoding input/output claims) and actors introduced in the simple protocol can be reused almost identically in the new setting.

If n is the number of processors, $2 \cdot n$ actor objects are required, since each processor may act both as sender—when running thread-code—or as receiver—when running NMI handler code. Though executed on the same processor, both code portions are two logically different entities, possibly residing in different protocol states, and owning different sets of mailboxes. That is also how we deal with thread and NMI handler concurrency: each of the NMI handler and the thread code own (and thus approve) separate actors. Note that in the IPI case, a single actor may communicate with many other partners, requiring it to maintain protocol state (the wait flag, and the remote and local input fields) per processor. The invariants of the manager are similar to those from the simple protocol.

C. Multiprocessor TLB Flush

The TLB abstraction and specification is similar to the previous section, but with a separate TLB for each processor.

D. IPIs in Microsoft's Hyper-V^{TM} Hypervisor

In the context of the Verisoft project we also studied the correctness of the IPI mechanism implemented in Microsoft's Hyper-V hypervisor. Though comparable in complexity to the IPI routine of the academic hypervisor, there are several differences:

- **Efficiency.** By introducing additional protocol variables sequential access to some of the shared data can be ensured, and thus fewer (costly) interlocked operations are required.
- **Lazy work.** The interrupt handler signals the receipt of the request and the accomplishing of the work separately. This allows for implementing less blocking caller code.

We have verified the implementation against a non-generic specification in VCC and are confident that this effort can be easily adapted to the generic specification used here.

VI. Conclusion

The verification presented here achieves the desired goal—it allows IPC clients to reason about IPCs like local procedure calls. As future work, the structure presented in Section III can be made modular even with respect to the set of functions provided via IPC. We can improve the structure slightly by changing the *Mgr* type; instead of the maps *InP* and *OutP*, the *Mgr* could hold a mapping of function tags to function objects, where each function object has its own *InP* and *OutP* maps. This would allow function objects to be reused in different managers, or even dynamically registered for IPC.

In principle, the technique presented here could also be applied to RPC, where the caller and callee execute in different address spaces. This requires translating the claims representing the pre- and post-conditions from one address space to the other. One possible way to achieve this effect would be to take the claim in the caller space, couple this to a second state in a way that captures the guarantees of the RPC, and existentially quantify away the caller space.

Acknowledgment

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft XT project under grant 01 IS 07 008. Work of the third author done while at DFKI GmbH, Saarbrücken, Germany.

References

[1] Eyad Alkassar, Sebastian Bogan, and Wolfgang Paul. Proving the correctness of client/server software. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34(1):145–191, 2009.

[2] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *POPL 2005*, pages 259–270. ACM, 2005.

[3] Manfred Broy, Stephan Merz, and Katharina Spies, editors. *Formal Systems Specification*, volume 1169 of *LNCS*. Springer, 1996.

[4] Ernie Cohen and Leslie Lamport. Reduction in TLA. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR 1998*, number 1466 in LNCS, pages 317–331. Springer, 1998.

[5] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV 2010*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.

[6] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *JAR*, 42(2–4):349–388, 2009.

[7] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

[8] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *JAR*, 42(2–4):301–347, 2009.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. seL4: Formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

[10] Microsoft Corp. VCC: A C Verifier. http://vcc.codeplex.com/, 2009.