VCC: A Practical System for Verifying Concurrent C

Ernie Cohen[1], Markus Dahlweid[2], Mark Hillebrand[3], Dirk Leinenbach[3], Michał Moskal[2], Thomas Santen[2], Wolfram Schulte[4], and Stephan Tobies[2]

[1] Microsoft Corporation, Redmond, WA, USA
ernie.cohen@microsoft.com
[2] European Microsoft Innovation Center, Aachen, Germany
{markus.dahlweid,michal.moskal,thomas.santen,stephan.tobies}@microsoft.com
[3] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
{mah,dirk.leinenbach}@dfki.de
[4] Microsoft Research, Redmond, WA, USA
schulte@microsoft.com

Abstract. VCC is an industrial-strength verification environment for low-level concurrent system code written in C. VCC takes a program (annotated with function contracts, state assertions, and type invariants) and attempts to prove the correctness of these annotations. It includes tools for monitoring proof attempts and constructing partial counterexample executions for failed proofs. This paper motivates VCC, describes our verification methodology, describes the architecture of VCC, and reports on our experience using VCC to verify the Microsoft Hyper-V hypervisor.[5]

1 Introduction

The mission of the Hypervisor Verification Project (part of Verisoft XT [1]) is to develop an industrially viable tool-supported process for the sound verification of functional correctness properties of commercial, off-the-shelf, system software, and to use this process to verify the Microsoft Hyper-V hypervisor. In this paper, we describe the proof methodology and tools developed in pursuit of this mission.

Our methodology and tool design has been driven by the following challenges:

Reasoning Engine. In an industrial process, developers and testers must drive the verification process (even if more specialized verification engineers architect global aspects of the verification, such as invariants on types). Thus, verification should be primarily driven by assertions stated at the level of code itself, rather than by guidance provided to interactive theorem provers. The need for mostly automatic reasoning led us to generate verification conditions that could be discharged automatically by an SMT (first-order satisfiability modulo theories) solver. The determination to stick to first-order methods means that the

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

only form of induction available is computational induction, which required developing methodological workarounds for inductive data structures.

Moreover, to allow users to understand failed verification attempts, we try whenever possible to reflect information from the underlying reasoning engine back to the level of the program. For example, countrexamples generated by failed proofs in the SMT solver are reflected to the user as (partial) counterexample traces by the VCC Model Viewer.

Weak Typing. Almost all critical system software today is written in C (or C++). C has only a weak, easily circumvented type system and explicit memory (de)allocation, so memory safety has to be explicitly verified. Moreover, address arithmetic enables many nasty programming tricks that are absent from typesafe code.

Still, most code in a well-written C system adheres to a much stricter type discipline. The VCC memory model [2] leverages this by maintaining in ghost memory a *typestate* that tracks where the "valid" typed memory objects are. On each memory reference and pointer dereference, there is an implicit assertion that resulting object is in the typestate. System invariants guarantee that valid objects do not overlap in any state, so valid objects behave like objects in a modern (typesafe) OO system. Well-behaved programs incur little additional annotation overhead, but nasty code (e.g., inside of the memory allocator) may require explicit manipulation of the typestate[6].

While C is flexible enough to be used in a very low-level way, we still want program annotations to take advantage of the meaningful structure provided by well-written code. Because C structs are commonly used to group semantically related data, we use them by default like objects in OO verification methodologies (e.g., as the container of invariants). Users can introduce additional (ghost) levels of structure to reflect additional semantic structure.

Concurrency. Most modern system software is concurrent. Indeed, the architecturally visible caching of modern processors means that even uniprocessor operating systems are effectively concurrent programs. Moreover, real system code makes heavy use of lock-free synchronization. However, typical modular and thread-local approaches to verifying concurrent programs (e.g., [3]) are based on locks or monitors, and forbid direct concurrent access to memory.

As in some other concurrency methodologies (e.g., [4]), we use an ownership discipline that allows a thread to perform sequential writes only to data that it owns, and sequential reads only to data that it owns or can prove is not changing. But in addition, we allow concurrent access to data that is marked as volatile in the typestate (using operations guaranteed by the compiler to be atomic on the given platform), leveraging the observation that a correct concurrent program typically can only race on volatile data (to prevent an optimizing compiler from

[6] Our slogan is "It's no harder to functionally verify a typesafe program written in an unsafe language than one written in a safe language." Thus, verification actually makes unsafe languages more attractive.

changing the program behavior). Volatile updates are required to preserve invariants but are otherwise unconstrained, and such updates do not have to be reported in the framing of a function specification.

Cross-Object Invariants. A challenge in modular verification is to how to make sure that updates don't break invariants that are out of scope. This is usually accomplished by restricting invariants to data within the object, or mild relaxations based on the ownership hierarchy. However, sophisticated implementations often require coordination between unrelated objects.

Instead, we allow invariants to mention arbitrary parts of the state. To keep modular verification sound, VCC checks that no object invariant can be broken by invariant-preserving changes to other objects. This *admissibility* check is done based on the type declarations alone.

Simulation. A typical way to prove properties of a concurrent data type is to show that it simulates some simpler type. In existing methodologies, simulation is typically expressed as a theorem about a program, e.g., by introducing an existentially quantified variable representing the simulated state. This is acceptable when verifying an abstract program (expressed, say, with a transition relation), but is awkward when updates are scattered throughout the codebase, and it violates our principle of keeping the annotations tightly integrated with the code.

Instead, we prove concurrent simulation in the code itself, by representing the abstract target with ghost state. The coupling invariant is expressed as an ordinary (single state) invariant linking the concrete and ghost state, and the specification of the simulated system is expressed with a two-state invariant on the ghost state. These invariants imply a (forward) simulation, with updates to the ghost state providing the needed existential witnesses.

Claims. Concurrent programs implicitly deal with chunks of knowledge about the state. For example, a program attempting to acquire a spin lock must "know" that the spin lock hasn't been destroyed. But such knowledge is ephemeral – it could be broken by any write that is not thread local – so passing knowledge to a function in the form of a precondition is too weak. Instead, we package the knowledge as the invariant of a ghost object; these knowledge-containing objects are called *claims*. Because claims are first-class objects, they can be passed in and out of functions and stored in data structures. They form a critical part of our verification methodology.

C and Assembly Code. System software requires interaction between C and assembly code. This involves subtleties such as the semantics of calls between C and assembly (in each direction), and consideration of which hardware resources (e.g., general purpose registers) can be silently manipulated by compiled C code. Assembly verification in VCC is discussed in [5].

Weak Memory Models. Concurrent program reasoning methods usually tacitly assume sequentially consistent memory. However, system software has to run on modern processors which, in a concurrent setting, do not provide an efficient implementation of sequentially consistent memory (primarily because of architecturally visible store buffering). Additional proof obligations are needed to guarantee that sequentially consistent reasoning is sound. We have developed a suitable set of conditions for x64 memory, but VCC does not yet enforce the corresponding verification conditions.

Content. Section 2 gives an overview of Hyper-V. Section 3 introduces the VCC methodology. Section 4 looks at the main components of VCC's tool suite. Section 5 reflects on the past year's experience on using VCC for verifying Hyper-V. Section 6 concludes with related work.

2 The Microsoft Hypervisor

The development of our verification environment is driven by the verification of the Microsoft Hyper-V hypervisor, which is an ongoing collaborative research project between the European Microsoft Innovation Center, Microsoft Research, the German Research Center for Artificial Intelligence, and Saarland University in the Verisoft XT project. The hypervisor is a relatively thin layer of software (100KLOC of C, 5KLOC of assembly) that runs directly on x64 hardware. The hypervisor turns a single real multi-processor x64 machine (with AMD SVM [6] or Intel VMX [7] virtualization extensions) into a number of virtual multi-processor x64 machines. (These virtual machines include additional machine instructions to create and manage other virtual machines.)

The hypervisor was not written with formal verification in mind. Verification requires substantial annotations to the code, but these annotations are structured so that they can be easily removed by macro preprocessing, so the annotated code can still be compiled by the standard C compiler. Our goal is that the annotations will eventually be integrated into the codebase and maintained by the software developers, evolving along with the code.

The hypervisor code consists of about 20 hierarchical layers, with essentially no up-calls except to pure functions. The functions and data of each layer is separated into public and private parts. These visibility properties are ensured statically using compiler and preprocessor hacks, but the soundness of the verification does not depend on these properties. These layers are divided into two strata. The lower layers form the *kernel stratum* which is a small multi-processor operating system, complete with hardware abstraction layer, kernel, memory manager, and scheduler (but no device drivers). The *virtualization stratum* runs in each thread an "application" that simulates an x64 machine without the virtualization features, but with some additional machine instructions, and running under an additional level of memory address translation (so that each machine can see 0-based, contiguous physical memory).

For the most part, a virtual machine is simulated by simply running the real hardware; the extra level of virtual address translation is accomplished by

```
typedef enum { Undefined, Initialized, Active, Terminating } LifeState;

typedef struct _Partition {
  bool signaled;
  LifeState lifeState;
  invariant(lifeState == Initialized || lifeState == Active ||
            lifeState == Terminating)
  invariant(signaled ==> lifeState == Active)
} Partition;

void part_send_signal(Partition *part)
  requires(part->lifeState == Active)
  ensures(part->signaled)
  maintains(wrapped(part))
  writes(part)
{
  unwrap(part);
  part->signaled = 1;
  wrap(part);
}
```

Listing 1. Sequential partition

using *shadow page tables* (SPTs). The SPTs, along with the hardware *translation lookaside buffers* (TLBs) (which asynchronously gather and cache virtual to physical address translations), implement a virtual TLB. This simulation is subtle for two reasons. First, the hardware TLB is architecturally visible, because (1) translations are not automatically flushed in response to edits to page tables stored in memory, and (2) translations are gathered asynchronously and nonatomically (requiring multiple reads and writes to traverse the page tables), creating races with system code that operates on the page tables. Even the semantics of TLBs are subtle, and the hypervisor verification required constructing the first accurate formal models of the x86/x64 TLBs. Second, the TLB simulation is the most important factor in system performance; simple SPT algorithms, even with substantial optimization, can introduce virtualization overheads of 50% or more for some workloads. The hypervisor therefore uses a very large and complex SPT algorithm, with dozens of tricky optimizations, many of which leverage the freedoms allowed by the weak TLB semantics.

3 VCC Methodology

VCC extends C with annotations giving function pre- and post-conditions, assertions, type invariants, and ghost code. Many of these annotations are similar to those found in ESC/Java [8], Spec# [9], or Havoc [10]. With contracts in place, VCC performs a static modular analysis, in which each function is verified in isolation, using only the contracts of functions that it calls and invariants of types used in its code. But unlike the aforementioned systems, VCC is geared towards sound verification of functional properties of low-level concurrent C code.

We show VCC's use by specifying hypervisor partitions; the data structure which keeps state to execute a guest operating system. Listing 1 shows a much simplified but annotated definition of the data structure. (The actual struct has 98 fields.)

Function Contracts. Every function can have a specification, consisting of four kinds of clauses. Preconditions, introduced by ***requires*** clauses, declare under which condition the function is allowed to be called. Postconditions, introduced by ***ensures*** clauses, declare under which condition the function is allowed to return. A ***maintains*** clause combines a precondition followed by a postcondition with the same predicate. Frame conditions, introduced by ***writes*** clauses, limit the part of the program's state that the function is allowed to modify.

So `part_send_signal` of Listing 1 is allowed to be called if the actual parameter's `lifeState` is `Active`. When the function terminates it guarantees (1) that the formal parameter's `signaled` bit has been set and (2) that it modified at most the passed partition object. We will discuss the notion of a wrapped object, which is mentioned in the ***maintains*** clause, in Sect. 3.1.

Type Invariants. Type definitions can have type invariants, which are one- or two-state predicates on data. Other specifications can refer to the invariant of object o as ***inv***(*o*) (or ***inv2***(*o*)). VCC implicitly uses invariants at various locations, as will be explained in the following subsections.

The invariant of the *`Partition`* struct of Listing 1 says that `lifeState` must be one of the valid ones defined for a partition, and that if the signaled bit is set, `lifeState` is `Active`.

Ghosts. A crucial concept in the VCC specification language is the division into operational code and ghost code. Ghost code is seen only by the static verifier, not the regular compiler. Ghost code comes in various forms: *Ghost type definitions* are types, which can either be regular C types, or special types for verification purposes like maps and claims (see Sect. 3.2). *Ghost fields* of arbitrary type can be introduced as specially marked fields in operational types. These fields do not interfere with the size and ordering of the operational fields. Likewise, static or automatic *ghost variables* of arbitrary type are supported. Like ghost fields, they are marked special and do not interfere with operational variables. *Ghost parameters* of arbitrary type can pass additional ghost state information in and out of the called function. *Ghost state updates* perform operations on only the ghost memory state. Any flow of data from the ghost state to the operational state of the software is forbidden.

One application of ghost code is maintaining *shadow copies* of implementation data of the operational software. Shadow copies usually introduce abstractions, e.g., representing a list as a set. They are also introduced to allow for atomic update of the shadow, even if the underlying data structure cannot be updated atomically. The atomic updates are required to enforce protocols on the overall system using two-state invariants.

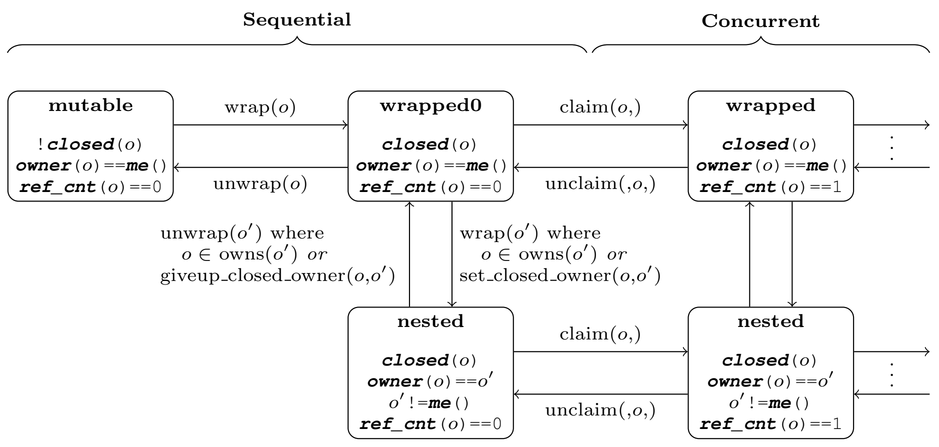

Fig. 1. Objects states, transitions, and access permissions

3.1 Verifying Sequential Programs

Ownership and Invariants. To deal with high-level aliasing, VCC implements a Spec#-style [9] ownership model: The heap is organized into a forest of tree structures. The edges of the trees indicate ownership, that is, an aggregate / subobject relation. The roots of trees in the ownership forest are objects representing threads of execution. The set of objects directly or transitively owned by an object is called the ownership domain of that object.

We couple ownership and type invariants. Intuitively, a type invariant can depend only on state in its ownership domain. We later relax this notion. Of course ownership relationships change over time and type invariants cannot always hold. We thus track the status for each object o in meta-state: **`owner`**`(o)` denotes the owner of an object, **`owns`**`(o)` specifies the set of objects owned by o (the methodology ensures that **`owner`**`()` and **`owns`**`()` stay in sync), **`closed`**`(o)` guarantees that o's invariant holds.

Figure 1 discusses the possible meta-states of an object (The *Concurrent* part of this figure will be explained in Sect. 3.2):

– **`mutable`**`(o)` holds if o is not closed and is owned by the current thread (henceforth called **`me`**). Allocated objects are always mutable and *fresh* (i.e., not previously present in the owns-set of the current thread).
– **`wrapped`**`(o)` holds if o is closed and owned by **`me`**. Non-volatile fields of wrapped objects cannot change. (**`wrapped0`**`(o)` abbreviates **`wrapped`**`(o)` and **`ref_cnt`**`(o)==0`).
– **`nested`**`(o)` holds if o is closed and owned by an object.

Ghost operations, like **`wrap`**, **`unwrap`**, etc. update the state as depicted in Fig. 1; note that unwrapping an object moves its owned object from nested to wrapped, wrapping the object moves them back.

```
typedef struct vcc(dynamic_owns) _PartitionDB {
  Partition *partitions[MAXPART];
  invariant(forall(unsigned i; i < MAXPART;
                    partitions[i] != NULL ==> set_in(partitions[i], owns(this))))
} PartitionDB;

void db_send_signal(PartitionDB *db, unsigned idx)
  requires(idx < MAXPART)
  maintains(wrapped(db))
  ensures((db->partitions[idx] != NULL) && (db->partitions[idx]->lifeState == Active)
          ==> db->partitions[idx]->signaled)
  writes(db)
{
  unwrap(db);
  if ((db->partitions[idx] != NULL) && (db->partitions[idx]->lifeState == Active)) {
    part_send_signal(db->partitions[idx]);
  }
  wrap(db);
}
```

Listing 2. Sequential partition database

The function `part_send_signal()` from Listing 1 respects this meta-state protocol. The function precondition requires `part` to be wrapped, i.e., `part`'s invariant holds. The function body first unwraps `part`, which suspends its invariant, next its fields are written to. To establish the postcondition, `part` is wrapped again; at this point, VCC checks that all invariants of `part` hold.

The `write` clauses work accordingly: write access to the root of an ownership domain enables writing to the entire ownership domain. In our example, ***`writes`***`(part)` gives the `part_send_signal` function write access to all fields of `part` (and the objects `part` owns), and tells the caller, that state updates are confined to the ownership domain of `part`. Additionally, one can always write to objects that are fresh.

Conditional Ownership. The actual hypervisor implementation does not use partition pointers but abstract partition identifiers to refer to partitions. This is because partitions can be created and destroyed anytime during the operation of the hypervisor, which might lead to dangling pointers. Listing 2) simulates the hypervisor solution: before any operation on a partition can take place, the pointer to the partition is retrieved from a partition database using the partition identifier. The *`PartitionDB`* type contains an array `partitions` of `MAXPART` entries of pointers to *`Partition`*. The index in `partitions` serves as the partition identifier. The partition database invariant states that all (non-null) elements of `partitions` are owned by the partition database.

The function `db_send_signal()` in Listing 2 attempts to send a signal to the partition with index `idx` of the partition database `db`. Is uses the function `part_send_signal()` from Listing 1, so we need to ensure that the preconditions of that function are met: `part->lifeState` is `Active` follows from the condition of the if-statement; ***`wrapped`***`(part)` holds since `part` is contained in the owns-set of `db`; unwrapping `db` transitions the partition from nested into the wrapped state. It also makes the partition writable as it has not previously been

```
#define ISSET(n, v) (((v) & (1ULL << (n))) != 0)
typedef Partition *PPartition;

typedef struct vcc(claimable) _PartitionDB {
  volatile uint64_t allSignaled;
  volatile PPartition partitions[MAXPART];
  invariant(forall(unsigned i; i < MAXPART;
                   unchanged(partitions[i]) ||
                   old(partitions[i]) == NULL || !closed(old(partitions[i]))))
  invariant(forall(unsigned i; i < MAXPART;
                   unchanged(ISSET(i, allSignaled)) ||
                   inv2(partitions[i])))
} PartitionDB;
```

Listing 3. Concurrent partition database

owned by the current thread and thus is also considered fresh from the point of view of the current thread.

3.2 Verifying Concurrent Programs

We now proceed with a concurrent version of the partition database structure from the previous example (cf. Listing 3). The array of partitions is declared as **volatile** to mark the intent of allowing arbitrary threads to add and remove partitions without unwrapping the partition database. The partitions are also no longer owned by the database, instead we imagine that the thread currently executing a partition owns it.[7] The first two-state invariant of the database prevents removal of closed partitions. The meaning of the invariant is: for any two consecutive states of the machine either `partitions[i]` stays the same (***unchanged***`(x)` is defined as ***old***`(x)==(x)`), or it was `NULL` in the first state, or the object pointed to by `partitions[i]` in the first state is open. VCC enforces this two-state invariant on every write to the database. Thus, if one has a closed partition at index `i`, one can rely on it staying there.

In the concurrent database, the individual `signaled` fields from the sequential version have been collected into a bit mask in the database. This allows taking an atomic snapshot of partitions currently being signaled. On the other hand, the details of how these bits can change logically belong with the individual partitions. This is stated by the second database invariant, saying that whenever the `i`-th bit of `allSignaled` is changed, the invariant of the `i`-th partition shall be preserved.

Listing 4 shows the updated version of the partition. The `lifeState` field remains the same. Since it is not marked **volatile**, the partition must be unwrapped before changing its life state. Because we want to keep the signature of the `part_send_signal()` function, the partition now needs to hold a pointer to the database and its index. An invariant enforces that the current partition

[7] In reality, if one takes a reference to a partition from the database, the database needs to provide some guarantees that the partition will stick around long enough. This is achieved using rundowns, but for brevity we skip it here.

```
typedef struct vcc(claimable) _Partition {
  LifeState lifeState;
  invariant(lifeState == Initialized || lifeState == Active ||
            lifeState == Terminating)

  struct _PartitionDB *db;
  unsigned idx;
  invariant(idx < MAXPART && db->partitions[idx] == this)

  spec(volatile bool signaled;)
  invariant(signaled <==> ISSET(idx, db->allSignaled))
  invariant(signaled ==> lifeState == Active)

  spec(claim_t db_claim;)
  invariant(keeps(db_claim) && claims_obj(db_claim, db))
} Partition;
```

Listing 4. Admissibility, volatile fields, shadow fields

is indeed stored at that index. This makes the invariant of the partition depend on fields of the database; so without further precaution, we would need to check the invariants of partitions when updating the database – but this would make reasoning about invariants non-modular. Instead, VCC requires that invariants are *admissible*, as described below.

Admissibility. A state transition is *legal* iff, for every object o that is closed in the transition's prestate or poststate, if any field of o is updated (including the "field" indicating closedness) the two-state invariant of o is preserved. An invariant of an object o is *admissible* iff it is satisfied by every legal state transition. Stated differently, an invariant is admissible if it is preserved by every transition that preserves invariants of all modified objects. Note that admissibility depends only on type definitions (not function specs or code), and is monotonic (i.e., if an invariant has been proved admissible, the addition of further types or invariants cannot make it inadmissible). VCC checks that all invariants are admissible. Thus, when checking that a state update doesn't break any invariant, VCC has to check only the invariants of the updated objects.

Some forms of invariants are trivially admissible. In particular, an invariant in object o that mentions only fields of o is admissible. This applies to `idx < MAXPART`. For `db->partitions[idx]==`***`this`***, let us assume that `db->partitions[idx]` changes across a transition (other components of that expression could not change). We know `db->partitions[idx]` was ***`this`*** in the prestate. Assume for a moment, that we know `db` was closed in both the prestate and the poststate. Then we know `db->partitions[idx]` was unchanged, it was `NULL` in the prestate (but ***`this`*** `!= NULL`), or ***`this`*** was open in the poststate: all three cases are contradictory. But if we knew that `db` stays closed, then the invariant would be admissible.

Claims. The required knowledge is provided by the *claim* object, owned by the partition and stored in the ghost field `db_claim`. A claim, as it is used here, can be thought of as a handle that keeps its claimed object from opening. If an object

```
void part_send_signal(Partition *part spec(claim_t c))
  requires(wrapped(c) && claims_obj(c, part))
{
  PartitionDB *db = part->db;
  uint64_t idx = part->idx;

  if (part->lifeState != Active) return;

  bv_lemma(forall(int i, j; uint64_t v; 0 <= i && i < 64 && 0 <= j && j < 64 ==>
    i != j ==> (ISSET(j, v) <==> ISSET(j, v | (1ULL << i)))));

  atomic(part, db, c) {
    speconly(part->signaled = true;)
    InterlockedBitSet(&db->allSignaled, idx);
  }
}
```

Listing 5. Atomic operation

o has a type which is marked with **`vcc(claimable)`** the field **`ref_cnt`**`(`o`)` tracks the number of outstanding claims that claim o. An object cannot be unwrapped if this count is positive, and a claim can only be created when the object is closed. Thus, when a claim to an object exists, the object is known to be closed.[8]

More generally, a claim is created by giving an invariant and a set of claimed (claimable) objects on which the claim depends. At the point at which the claim is created, the claimed objects must all be closed and the invariant must hold. Moreover, the claim invariant, conjoined with the invariants of the claimed objects, must imply that the claim invariant cannot be falsified without opening one of the claimed objects.

Pointers to claims are often passed as ghost arguments to functions (most often with the precondition that the claim is wrapped). In this capacity, the claim serves as a stronger kind of precondition. Whereas an ordinary precondition can only usefully constrain state local to the thread, a claim can constrain volatile (shared) state. Moreover, unlike a precondition, the claim invariant is guaranteed to hold until the claim is destroyed. In a function specification, the macro **`always`**`(`o, P`)` means that the function maintains **`wrapped`**`(`o`)` and that o points to a valid claim, and that the invariant of the claim o implies the predicate P. Thus, this contract guarantees that P holds throughout the function call (both to the function and to its callers).

Atomic Blocks. Listing 5 shows how objects can be concurrently updated. The signaling function now only needs a claim to the partition, passed as a ghost parameter, and does not need to list the partition in its writes clause. In fact, the writes clause of the signaling function is empty, reflecting the fact that from the caller perspective, the actions could have been performed by another thread, without the current thread calling any function. A thread can read its own non-volatile state; it can also read non-volatile fields of closed objects, in particular objects for which it holds claims. On the other hand, the volatile fields can

[8] Claims can actually be implemented using admissible two-state invariants. We decided to build them into the annotation language for convenience.

only be read and written inside of atomic blocks. Such a block identifies the objects that will be read or written, as well as claims that are needed to establish closedness of those objects. It can contain at most one physical state update or read, which is assumed to be performed atomically by the underlying hardware. In our example, we set the `idx`-th bit of `allSignaled` field, using a dedicated CPU instruction (it also returns the old value, but we ignore it). On top of that, the atomic block can perform any number of updates of the ghost state. Both physical and ghost updates can only be performed on objects listed in the header of the atomic block. The resulting state transition is checked for legality, i.e., we check the two-state invariants of updated objects across the atomic block. The beginning of the atomic block is the only place where we simulate actions of other threads; technically this is done by forgetting everything we used to know about volatile state. The only other possible state updates are performed on mutable (and thus open) objects and thus are automatically legal.

3.3 Verification of Concurrency Primitives

In VCC, concurrency primitives (other than atomic operations) are verified (or just specified), rather than being built in. As an example we present the acquisition of a reader-writer lock in exclusive (i.e., writing) mode.[9] In this example, claims are used to capture not only closedness of objects but also properties of their fields.

The data structure *`LOCK`* (cf. Listing 6) contains a single volatile implementation variable called `state`. Its most significant bit holds the write flag that is set when a client requests exclusive access. The remaining bits hold the number of readers. Both values can be updated atomically using interlocked operations.

Acquiring a lock in exclusive mode proceeds in two phases. First, we spin on setting the write bit of the lock atomically. After the write bit has been set, no new shared locks may be taken. Second, we spin until the number of readers reaches zero. This protocol is formalized using lock ghost fields and invariants.

The lock contains four ghost variables: a pointer `protected_obj` identifying the object protected by the lock, a flag `initialized` that is set after initialization, a flag `writing` that is one when exclusive access has been granted (and no reader holds the lock), and a claim `self_claim`. The use of `self_claim` is twofold. First, we tie its reference count to the implementation variables of the lock. This allows restricting changes of these variables by maintaining claims on `self_claim`. Second, it is used to claim lock properties, serving as a proxy between the claimant and the lock. For this purpose it claims the lock and is owned by it. It thus becomes writable and claimable in atomic operations on the lock without requiring it or the lock to be listed in function writes clauses.

Figures 2 and 3 contain a graphical representation of the lock invariants. Figure 2 shows the setup of ownership and claims. The lock access claim is created after initialization. It ensures that the lock remains initialized and allocated,

[9] For details and full annotated source code see [11].

```
#define Write(state) ((state)&0x80000000)
#define Readers(state) ((state)&0x7FFFFFFF)

typedef struct vcc(claimable) vcc(volatile_owns) _LOCK {
  volatile long state;
  spec(obj_t protected_obj;)
  spec(volatile bool initialized, writing;)
  spec(volatile claim_t self_claim;)

  invariant(old(initialized) ==> initialized && unchanged(self_claim))
  invariant(initialized ==>
    is_claimable(protected_obj) && is_non_primitive_ptr(protected_obj) &&
    set_in(self_claim,owns(this)) && claims_obj(self_claim, this) &&
    protected_obj != self_claim)
  invariant(initialized && !writing ==>
    set_in(protected_obj,owns(this)) &&
    ref_cnt(protected_obj) == (unsigned) Readers(state) &&
    ref_cnt(self_claim) == (unsigned)(Readers(state) + (Write(state)!=0)))
  invariant(initialized && old(Write(state)) ==>
    Readers(state) <= old(Readers(state)) && (!Write(state) ==> old(writing)))
  invariant(initialized && writing ==>
    Readers(state) == 0 && Write(state) && ref_cnt(self_claim) == 0)
} LOCK;
```

Listing 6. Annotated lock data structure

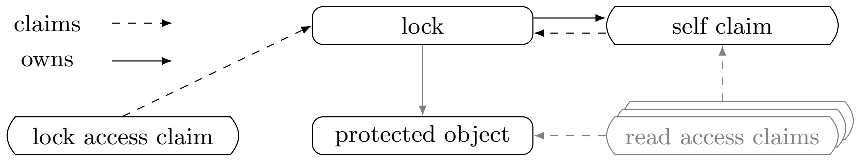

Fig. 2. Ownership and claims structure (shared and exclusive access)

and clients use it (or a derived claim) when calling lock functions. During non-exclusive access each reader holds a read access claim on the protected object and the lock, and the lock owns the protected object, as indicated in gray. During exclusive access the protected object is owned by the client and there may be no readers. Figure 3 depicts the dynamic relation between implementation and ghost variables. As long as the write bit is zero, shared locks may be acquired and released, as indicated by the number of readers. The write bit is set when the acquisition of an exclusive lock starts. In this phase the number of readers must decrease. When it reaches zero, exclusive lock acquisition can complete by activating the `writing` flag. For each reader and each request for write access (which is at most one) there is a reference on `self_claim`.

Listing 7 shows the annotated code for acquisition of an exclusive lock. The macro ***claimp*** wrapped around the parameter `lock_access_claim` means that `lock_access_claim` is a ghost pointer to a wrapped claim; the ***always*** clause says that this claim is wrapped, is not destroyed by the function, and that its invariant implies that the lock is closed and initialized (and hence, will remain so during the function call). After the function returns it guarantees that the protected object is unreferenced, wrapped, and fresh (and thus, writable).

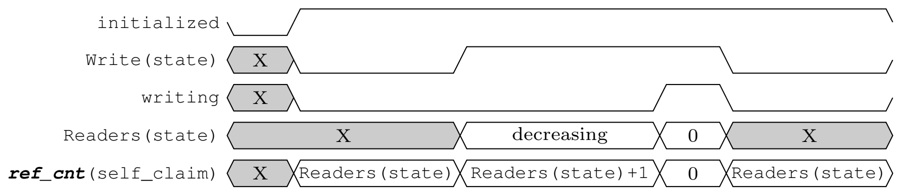

Fig. 3. Relation of lock implementation and ghost variables

In the first loop of the implementation we spin until the write bit could be atomically set (via the `InterlockedOr` intrinsic), i.e., in an atomic block the write bit has been seen as zero and then set to one. In the terminating loop case we create a temporary claim `write_bit_claim`, which references the self claim and states that the lock stays initialized, that the self claim stays, and that the write bit of the lock has been set. VCC checks that the claimed property holds initially and is stable against interference. The former is true by the passed-in lock access claim and the state seen and updated in the atomic operation; the latter is true because as long as there remains a reference to the self claim, the `writing` flag cannot be activated and the write bit cannot be reset. Also, the atomic update satisfies the lock invariant.

The second loop waits for the readers to disappear. If the number of readers has been seen as zero, we remove the protected object from the ownership of the lock, discard the temporary claim, and set the `writing` flag. All of this can be justified by the claimed property and the lock's invariant. Setting the `writing` flag is allowed because the write bit is known to be set. Furthermore, the `writing` flag is known to be zero in the pre-state of the atomic operation because the reference count of the self claim, which is referenced by `write_bit_claim`, cannot be zero. This justifies the remaining operations.

4 VCC Tool Suite

VCC reuses the Spec# tool chain [9], which has allowed developing a comprehensive C verifier with limited effort. In addition we developed auxiliary tools to support the process of verification engineering in a real-world effort.

4.1 The Static Verifier

We base our verification methodology on inline annotations in the, otherwise unaltered, source code of the implementation. The C preprocessor is used to eliminate these annotations for normal C compilation. For verification, the output of the preprocessor (with annotations still intact) is fed to the VCC compiler.

CCI. The VCC compiler is build using Microsoft Research's Common Compiler Infrastructure (CCI) libraries [12]. VCC reads annotated C and turns the input into CCI's internal representation to perform name resolution, type and error check as any normal C compiler would do.

```
void AcquireExclusive(LOCK *lock claimp(lock_access_claim))
  always(lock_access_claim, closed(lock) && lock->initialized)
  ensures(wrapped0(lock->protected_obj) && is_fresh(lock->protected_obj))
{
  bool done;
  spec(claim_t write_bit_claim;)

  bv_lemma(forall(long i; Write(i|0x80000000) &&
    Readers(i|0x80000000) == Readers(i)));

  do
    atomic (lock, lock_access_claim) {
      done = !Write(InterlockedOr(&lock->state, 0x80000000));
      speconly(if (done) {
        write_bit_claim = claim(lock->self_claim, lock->initialized &&
          stays_unchanged(lock->self_claim) && Write(lock->state));
      })
    }
  while (!done);
  do
    invariant(wrapped0(write_bit_claim))
    atomic (lock, write_bit_claim) {
      done = Readers(lock->state)==0;
      speconly(if (done) {
        giveup_closed_owner(lock->protected_obj, lock);
        unclaim(write_bit_claim, lock->self_claim);
        lock->writing = 1;
      })
    }
  while (!done);
}
```

Listing 7. Acquisition of an exclusive lock

Source Transformations and Plugins. Next, the fully resolved input program undergoes multiple source-to-source transformations. These transformations first simplify the source, and then add proof obligations stemming from the methodology. The last transformation generates the Boogie source.

VCC provides a plugin interface, where users can insert and remove transformations, including the final translation. Currently two plugins have been implemented: to generate contracts for assembly functions from their C correspondants; and to build a new methodology based on separation logic [13].

Boogie. Once the source code has been analyzed and found to be valid, it is translated into a Boogie program that encodes the input program according to our formalization of C. Boogie [14] is an intermediate language that is used by a number of software verification tools including Spec# and HAVOC. Boogie adds minimal imperative control flow, procedural and functional abstractions, and types on top of first order predicated logic. The translation from annotated C to Boogie encodes both static information about the input program, like types and their invariants, and dynamic information like the control flow of the program and the corresponding state updates. Additionally, a fixed axiomatization of C memory, object ownership, type state, and arithmetic operations (the *prelude*) is added. The resulting program is fed to the Boogie program verifier, which translates it into a sequence of verification conditions. Usually, these are then passed to an automated theorem prover to be proved or refuted. Alternatively, they can be discharged interactively. The HOL-Boogie tool [15] provides support for this approach based on the Isabelle interactive theorem prover.

Z3. Our use of Boogie targets Z3 [16], a state-of-the art first order theorem prover that supports satisfiability modulo theories (SMT). VCC makes heavy use of Z3's fast decision procedures for linear arithmetic and uses the slower fixed-length bit vector arithmetic only when explicitely invoked by VCC's ***`bv_lemma`***`()` mechanism (see Listing 5 for an example). These lemmas are typically used when reasoning for overflowing arithmetic or bitwise operations.

4.2 Static Debugging

In the ideal case, the work flow described above is all there is to running VCC: an annotated program is translated via Boogie into a sequence of verification conditions that are successfully proved by Z3. Unfortunately, this ideal situation is encountered only seldomly during the process of verification engineering, where most time is spent debugging failed verification attempts. Due to the undecidability of the underlying problem, these failures can either be caused by a genuine error in either the code or the annotations, or by the inability of the SMT solver to prove or refute a verification condition within available resources like computer memory, time, or verification engineer's patience.

VCC Model Viewer. In case of a refutation, Z3 constructs a counterexample that VCC and Boogie can tie back to a location in the original source code. However that is not that easy, since these counterexamples contain many artifacts of the underlying axiomatization, and so are not well-suited for direct inspection. The VCC Model Viewer translates the counterexample into a representation that allows inspecting the sequence of program states that led to the failure, including the value of local and global variables and the heap state.

Z3 Inspector. A different kind of verification failure occurs when the prover takes an excessive amount of time to come up with a proof or refutation for a verification condition. To counter this problem, we provide the Z3 Inspector, a tool that allows to monitor the progress of Z3 tied to the annotations in the source code. This allows to pinpoint those verification conditions that take excessively long to be processed. There can be two causes for this: either the verification condition is valid and the prover requires a long time to find the proof, or the verification condition is invalid and the search for a counterexample takes a very long time. In the latter case, the Z3 Inspector helps identifying the problematic assertion quickly.

Z3 Axiom Profiler. In the former case a closer inspection of the quantifier instantiation pattern can help to determine inefficiencies in the underlying axiomatization of C or the program annotations. This is facilitated by the Z3 Axiom Profiler, which allows to analyze the quantifier instantiation patters to detect, e.g., triggering cycles.

Visual Studio. All of this functionality is directly accessible from within the Visual Studio IDE, including verifying only individual functions. We have found that this combination of tools enables the verification engineer to efficiently develop and debug the annotations required to prove correctness of the scrutinized codebase.

5 VCC Experience

The methodology presented in this paper was implemented in VCC in late 2008. Since this methodology differs significantly from earlier approaches, the annotation of the hypervisor codebase had to start from scratch. As of June 2009, fourteen verification engineers are working on annotating the codebase and verifying functions. Since November 2008 approx. 13 500 lines of annotations have been added to the hypervisor codebase. About 350 functions have been successfully verified resulting in an average of two verified functions per day. Additionally, invariants for most public and private data types (consisting of about 150 structures or groups) have been specified and proved admissable. This means that currently about 20% of the hypervisor codebase has been successfully verified using our methodology.

A major milestone in the verification effort is having the specifications of all public functions from all layers so that the verification of the different layers require no interaction of the verification engineers, since all required information has been captured in the contracts and invariants. This milestone has been reached or will be reached soon for seven modules. Also for three modules already more than 50% of the functions have been successfully verified.

We have found that having acceptable turnaround times for verify-and-fix cycles is crucial to maintain productivity of the verification engineers. Currently VCC verifies most functions in 0.5 to 500 seconds with an average of about 25 seconds. The longest running function needs ca. 2 000 seconds to be verified.

The all-time high was around 50 000 seconds for a successful proof attempt. In general failing proof attempts tend to take longer than successfully verifying a function. A dedicated test suite has been created to constantly monitor verification performance. Performance has improved by one to two orders of magnitude. Many changes have contributed to these improvements, ranging from changes in our methodology, the encoding of type state, our approach to invariant checking, the support of out parameters, to updates in the underlying tools Boogie and Z3. With these changes, we have, for example, reduced the verification time for the 50 000s function down to under 1 000s.

Still, in many cases the verification performance is unacceptable. Empirically, we have found that verification times of over a minute start having an impact on the productivity of the verification engineer, and that functions that require one hour or longer are essential intractable. We are currently working on many levels to alleviate these problems: improvements in the methodology, grid-style distribution of verification condition checking, parallelization of proof search for a single verification condition, and other improvements of SMT solver technology.

6 Related Work

Methodology. The approach of Owicki and Gries [17] requires annotations to be stable with respect to *every* atomic action of the other threads, i.e., that the assertions are interference free. This dependency on the other threads makes the Owicki-Gries method non-modular and the number of interference tests grows quadratically with the number of atomic actions. Ashcroft [18] proposed to just use a single big state invariant to verify concurrent programs. This gets rid of the interference check, and makes verification thread-modular.

Jones developed the more modular rely/guarantee method [19] which abstracts the possible interleavings with other threads to rely on and guarantee assertions. Now, it suffices to check that each thread respects these assertions locally and that the rely and guarantee assertions of all threads fit together. Still, their approach (and also the original approach of Owicki and Gries) do not support data modularity: there is no hiding mechanism, a single bit change requires the guarantees of *all* threads to be checked.

Flanagan et al. [3] describe a rely/guarantee based prover for multi-threaded Java programs. They present the verification of three synchronization primitives but do not report on larger verification examples. The approach is thread modular (as it is based on rely/guarantee) but not function modular (they simulate function calls by inlining).

In contrast to rely/guarantee, concurrent separation logic exploits that large portions of the program state may be operated on mutually exclusive [20, 21]. Thus, like in our approach, interference is restricted to critical regions and verification can be completely sequential elsewhere. Originally, concurrent separation logic was restricted to exclusive access (and atomic update) of shared resources. Later, Bornat proposed a fractional ownership scheme to allow for shared read-only state also [22]. Recently, Vafeiadis and Parkinson [23] have worked on combining the ideas of concurrent separation logic with rely/guarantee reasoning.

Our ownership model, with uniform treatment of objects and threads, is very similar to the one employed in Concurrent Spec# [4]. Consequently, the visible specifications of locks, being the basis of Concurrent Spec# methodology, is essentially the same. We however do not treat locks as primitives, and allow for verifying implementation of various concurrency primitives. The Spec# ideas have also permeated into recent work by Leino and Mueller [24]. They use dynamic frames and fractional permissions for verifying fine grained locking. History invariants [25] are two-state object invariants, requiring admissibility check similar to ours. These invariants are however restricted to be transitive and are only used in the sequential context.

Systems Verification. Klein [26] provides a comprehensive overview of the history and current state of the art in operating systems verification, which is supplemented by a recent special issue of the Journal of Automated Reasoning on operating system verification [27]. The VFiasco [28] project, followed by the Robin projects attempted the verification of a micro kernel, based on a translation of C++ code into its corresponding semantics in the theorem prover PVS. While these projects have been successful in providing a semantic model for C++, no significant portions of the kernel implementation has been verified. Recent related projects in this area include the project L4.verified [29] (verification of an industrial microkernel), the FLINT project [30] (verification of an assembly kernel), and the Verisoft project [31] (the predecessor project of Verisoft XT focusing on the pervasive verification of hardware-software systems). All three projects are based on interactive theorem proving (with Isabelle or Coq). Our hypervisor verification attempt is significantly more ambitious, both with respect to size (ca. 100KLOC of C) and complexity (industrial code for a modern multiprocessor architecture with a weak memory model).

Acknowledgments Thanks to everyone in the project: Artem Alekhin, Eyad Alkassar, Mike Barnett, Nikolaj Bjørner, Sebastian Bogan, Sascha Böhme, Matko Botinĉan, Vladimir Boyarinov, Ulan Degenbaev, Lieven Desmet, Sebastian Fillinger, Tom In der Rieden, Bruno Langenstein, K. Rustan M. Leino, Wolfgang Manousek, Stefan Maus, Leonardo de Moura, Andreas Nonnengart, Steven Obua, Wolfgang Paul, Hristo Pentchev, Elena Petrova, Norbert Schirmer, Sabine Schmaltz, Peter-Michael Seidel, Andrey Shadrin, Alexandra Tsyban, Sergey Tverdyshev, Herman Venter, and Burkhart Wolff.

References

1. Verisoft XT: The Verisoft XT project. `http://www.verisoftxt.de` (2007)
2. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: SSV 2009. ENTCS, Elsevier Science B.V. (2009)
3. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In Métayer, D.L., ed.: ESOP 2002. Number 2305 in LNCS, Springer (2002) 262–277
4. Jacobs, B., Piessens, F., Leino, K.R.M., Schulte, W.: Safe concurrency for aggregate objects with invariants. In Aichernig, B.K., Beckert, B., eds.: SEFM 2005, IEEE (2005) 137–147
5. Maus, S., Moskal, M., Schulte, W.: Vx86: x86 assembler simulated in C powered by automated theorem proving. In Meseguer, J., Roşu, G., eds.: AMAST 2008. Number 5140 in LNCS, Springer (2008) 284–298
6. Advanced Micro Devices (AMD), Inc.: AMD64 Architecture Programmer's Manual: Volumes 1–3. (2006)
7. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual: Volumes 1–3b. (2006)
8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. SIGPLAN Notices **37**(5) (2002) 234–245
9. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: CASSIS 2004. Number 3362 in LNCS, Springer (2004) 49–69
10. Microsoft Research: The HAVOC property checker. `http://research.microsoft.com/projects/havoc`
11. Hillebrand, M.A., Leinenbach, D.C.: Formal verification of a reader-writer lock implementation in C. In: SSV 2009. ENTCS, Elsevier Science B.V. (2009) Source code available at `http://www.verisoftxt.de/PublicationPage.html`.

12. Microsoft Research: Common compiler infrastructure. `http://ccimetadata.codeplex.com/`
13. Botinĉan, M., Parkinson, M., Schulte, W.: Separation logic verification of C programs with an SMT solver. In: SSV 2009. ENTCS, Elsevier Science B.V. (2009)
14. Barnett, M., Chang, B.Y.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO 2005. Number 4111 in LNCS, Springer (2006) 364–387
15. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie: An interactive prover-backend for the Verifiying C Compiler. Journal of Automated Reasoning (2009) To appear.
16. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In Ramakrishnan, C.R., Rehof, J., eds.: TACAS 2008. Number 4963 in LNCS, Springer (2008) 337–340
17. Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM **19**(5) (1976) 279–285
18. Ashcroft, E.A.: Proving assertions about parallel programs. Journal of Computer and System Sciences **10**(1) (1975) 110–135
19. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems **5**(4) (1983) 596–619
20. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science **375**(1-3) (2007) 271–307
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, IEEE (2002) 55–74
22. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In Palsberg, J., Abadi, M., eds.: POPL 2005, ACM (2005) 259–270
23. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In Caires, L., Vasconcelos, V.T., eds.: CONCUR 2007. Number 4703 in LNCS, Springer (2007) 256–271
24. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In Castagna, G., ed.: ESOP 2009. Volume 5502 of LNCS., Springer (2009) 378–393
25. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In Nicola, R.D., ed.: ESOP 2007. Number 4421 in LNCS, Springer (2007) 80–94
26. Klein, G.: Operating system verification – An overview. Sādhanā: Academy Proceedings in Engineering Sciences **34**(1) (February 2009) 27–69
27. Journal of Automated Reasoning: Operating System Verification **42**(2–4) (2009)
28. Hohmuth, M., Tews, H.: The VFiasco approach for a verified operating system. In: 2nd ECOOP Workshop in Programming Languages and Operating Systems. (2005)
29. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: Taking microkernels to the next level. SIGOPS Oper. Syst. Rev. **41**(4) (2007) 3–11
30. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In Schneider, K., Brandt, J., eds.: TPHOLs 2007, LNCS (2007) 189–206
31. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load: Leveraging a semantics stack for systems verification. In *Journal of Automated Reasoning: Operating System Verification* [27] 389–454