A Practical Verification Methodology for Concurrent Programs

February 12, 2009

Technical Report
MSR-TR-2009-15

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

A Practical Verification Methodology for Concurrent Programs

Ernie Cohen
Microsoft Corp.
ernie.cohen@microsoft.com

Michał Moskal
European Microsoft Innovation Center
michal.moskal@microsoft.com

Wolfram Schulte
Microsoft Research, Redmond
wolfram.schulte@microsoft.com

Stephan Tobies
European Microsoft Innovation Center
stephan.tobies@microsoft.com

Abstract

We describe a methodology for reasoning about realistic concurrent programs. Our methodology allows two-state invariants that span multiple objects without sacrificing thread- or data-modularity, as well as the derived construction of first-class objects that capture knowledge about the system state. The methodology has been implemented in an automatic sound verifier for concurrent C programs being used to verify the code of the Microsoft Hypervisor, the virtualization kernel of Hyper-V.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms concurrency, verification

Keywords concurrency, verification

1. Introduction

Despite significant advances over the last 30 years, there is still no generally applicable[1] practical methodology for verifying concurrent software. One reason for this is the tension between the need for low-level, fine-grained interaction between concurrent components (typically addressed with temporal logic and rely-guarantee (Jones 1983)) and the practical need to hide such interactions when specifying higher-level procedures (typically addressed in sequential code with object-oriented methods such as object invariants and ownership (Barnett et al. 2004)).

Attempts to extend sequential verification with object invariants to allow concurrency have generally followed the monitor approach (Hoare 1974). Here, the state is partitioned into a number of objects (or resources), each guarded by a lock. A thread can operate only on those objects that it owns. Acquiring a lock on an object transfers ownership of the object to the thread, bringing in the object invariant as an assumption; releasing the lock asserts the object invariant and removes the object from the threads ownership domain. Some examples of this tradition are concurrent extensions to ownership methodologies (Jacobs et al. 2007) and concurrent separation logic (CSL) (OHearn 2007a)[2]. In the monitor tradition, an atomic action is just sugar for code that acquires all needed locks, makes the updates, and releases the locks, where this code just happens to be compressed into a single atomic action (Parkinson et al. 2007). (For objects that are used only in this way, the locks are guaranteed to always be unlocked, and so can be removed from the implementation.)

The main challenge of the monitor approach is the design of the object invariants. Because an object invariant is checked only when the object is unlocked, an object invariant cannot mention the state of other objects without some restrictions (since a change to the state of the other object might break the invariant). For example, in CSL, object invariants cannot mention the states of other objects; in systems based on hierarchical ownership, the invariant of an object can typically depend only on the state of objects in the object's ownership domain (the objects transitively owned by the object). These limitations can make it difficult to prove functional correctness of interesting programs without growing the objects in ways that violate the natural encapsulation boundaries, making the method less practical as programs grow. (Some additional flexibility can be obtained through the use of read permissions (Bornat et al. 2005); see section 11).

As an example of this problem, consider a compiler running as a service inside of a development environment. The syntax tree nodes all share a symbol table to resolve symbolic identifiers to names. The nodes need the guarantee that this resolution will never fail. At the same time, the symbol table has to be able to change as additional identifiers are added to the program text (potentially concurrently, when different compiler threads provide services for multiple editing windows)[3].

Our methodology allows the specification of a two-state invariant that states that the symbol table can only grow. Two-state invariants restrict possible transitions of an object by specifying the relationship between the state before and after updating the object. In concrete syntax the pre- and post-state are referred to using the `old(...)` qualifier and unqualified expressions respectively. An

[1] By generally applicable, we mean a methodology that can be applied without rewriting the code. For example, it shouldn't depend on a particular programming discipline regarding how programs are synchronized. There are, of course, many useful classes of programs for which concurrent verification is easily reduced to sequential verification.

[2] For the current purpose, we can think of the objects of a CSL program as its resources; the memory associated with each can change, but they are always pairwise disjoint.

[3] This is just an easy to explain instance of a problem of the kind where multiple objects depend on a single context object.

invariant that does not use **old**(...) is effectively a single-state invariant that is supposed to hold after any update to the object.

For example, we might define symbol tables and expressions as follows [4]:

```
typedef struct {
  char *names[MAX_SYM];
  invariant(
    forall(uint i; old(names[i]) != NULL ==>
             old(names[i]) == names[i]))
} SYMBOL_TABLE;

typedef struct {
  int id;
  SYMBOL_TABLE *s;
  invariant(s->names[id] != NULL)
} EXPR;
```

Every transition of the entire system should fulfill the two-state invariants of all objects. Yet we want to reason about the code in a modular way and check the invariant of an object only when updating that very object. Additional restrictions on invariants are required to guarantee soundness of this approach. Particularly, the invariant of any object has to be preserved when other objects change (provided these changes observe the respective object's invariants). This condition, which we require on all invariants, is called *admissibility* of invariants.

Going back to our example, the invariant of SYMBOL_TABLE depends only on its own fields and thus is obviously preserved under changes to other objects. However the invariant of EXPR does rely on a field of the SYMBOL_TABLE. Yet if the SYMBOL_TABLE changes, it needs to preserve its invariant (only add elements), therefore it cannot invalidate the invariant of EXPR. Without the invariant on the symbol table, the invariant of EXPR would clearly be inadmissible.

Typically, object invariants cannot hold at all times; particularly during initialization, finalization and during non-atomic updates. To capture this common pattern we use the notion of *closed objects*, which is akin to the inv field in the Boogie methodology (Barnett et al. 2004). Two-state invariants need to hold only if the object is closed in either of the two states, so that an invariant can be disabled as required by opening the objects (assuming that the invariant allows the object to be opened up).

While an object invariant might not hold at all times, there has to be a way to enforce that an object stays closed. Otherwise, no object could rely on other objects' invariants to hold. The usual way to keep objects closed is via ownership: objects are owned by other objects and the (unique) owner controls opening and closing of the owned object. While often useful for sequential code, this is not appropriate for our example — for the invariant of EXPR to be admissible we need the symbol table to be closed, however at most one EXPR node can own the symbol table.

To solve this problem we introduce the notion of *claims*. A claim is a reified guarantee that its claimed object will stay closed as long as the claim is closed. By owning a claim, the owner of that claim has the guarantee that the claimed object will not open, regardless of the actual owner of the claimed object. Interestingly, claims do not need to be added as a language primitive; they can be implemented as ordinary objects and a two-state invariant on the claimed object restricting when the claimed object can be opened. To make the invariant of EXPR admissible we make it own a claim on the symbol table.

Summing up, the main contributions of this paper are as follows:

- We propose the use of **two-state object invariants**, which allow for the natural specification of atomic updates and lock-free data structures (like spinlocks, see Section 2, or lock-free stacks) and additional specification-related concepts like the aforementioned claims.
- We define invariant **admissibility**, which allows data-modular reasoning in the presence of cross-object invariants, thus enabling the specification and verification of a larger class of programs without breaking their natural encapsulation boundaries.
- We introduce **claims** as a way to reify interlinked knowledge about system state. Claims allow the construction of admissible invariants for many scenarios, where ordinary, tree-shaped ownership hierarchies are inadequate. Claims provide a flexible generalization of the concept of read permissions.
- Our methodology has been **implemented** in an automated sound verifier for concurrent C programs. It reads annotated C code, and uses Boogie (Barnett et al. 2006) to generate verification conditions that are subsequently discharged by the automated theorem prover Z3 (de Moura and Bjrner 2008). Using this implementation, we are applying our methodology to verification of Microsoft Hypervisor, the virtualization kernel of Microsoft's Hyper-V product.

2. An Overview of the Program Methodology

In the following, we present our methodology in more detail.

2.1 Objects and Invariants

As is usual in modular reasoning methods, we imagine a global state composed of a number of disjoint objects, the state of each given by a collection of fields. Objects may be created and destroyed. Some of the objects may be *ghost* (a.k.a. specification) objects that do not exist in the implementation but are included to facilitate program reasoning, and objects may also have ghost fields. (The means by which we overlay an object model on top of C goes beyond the scope of this paper, but is described in a companion paper.)

The verification condition we are generating states that a program with additional ghost state, executing under a restricted scheduler never "goes wrong", that is makes a transition that violates two-state invariant of any closed object. However we prove that those restricted executions reach the same set of states as the executions without ghost state under an unrestricted scheduler. Thus we can prove arbitrary properties of the real program by just making it "go wrong" when an undesirable state is detected. We return to this issue in Section 2.6.

Call a transition *safe* if it preserves all object invariants, and call a state safe if the stuttering transition (i.e., a transition from σ to σ) from that state satisfies all object invariants. We assume the initial state to be safe, by stating that all objects are initially open.

To keep reasoning modular, we would like to check (for each program transition) only the invariants of those objects whose state changes in the transition; as mentioned above, this is potentially unsound if object invariants can mention the states of other objects. Given a collection of invariants for each object, an object invariant is *admissible* iff it is preserved by every transition that preserves invariants of all modified objects. Note that admissibility does not depend on the program code.

[4] The utility of two-state invariants goes far beyond examples like this. In particular, a ghost object with a two-state invariant can be viewed as a temporal specification. We can prove that a concrete object simulates this specification by coupling the two with an ordinary invariant (in the concrete object); the ghost updates to the specification then serve to witness the simulation. Thus, instead of having to conduct simulation proofs externally, the combination of two-state invariants and ghost data lets us carry out such proofs using ordinary assertional verification techniques. The utility of simulation techniques in proving system properties is well-established (Lamport 2002).

As in other object invariant methodologies, we assume that each object has a Boolean ghost field closed that indicates when its invariant is expected to hold. Because we are using two-state invariants, the object invariant is required to hold between two successive states only if the object is closed in at least one of them. Closing/opening an object is akin to initializing/finalizing a state machine.

Fields of an object come in two flavors, *volatile* and *sequential* (the default). The value of a sequential field can change only when the object is open, while volatile fields can change at any time (subject to the object invariants when the object is closed)[5]. Thus, for any sequential field, we can think of the object as having an additional invariant to this effect.

2.2 Ownership

As in most approaches to thread-local verification, we start from disjoint concurrency, i.e. threads operating on disjoint portions of the state; this allows ordinary, sequential reasoning within a thread. Thus, in any state, each thread "owns" some portion of the state which it is allowed to read and write; inter-thread communication thus requires some transfer of owned state between threads. In most concurrent methodologies, this happens by transferring ownership via some *built-in* type of shared object, such as a resource or a lock; we use ordinary objects for this purpose. Indeed, we will show below an example how locks can be implemented and verified in our system.

Each object has a ghost field owns that maintains the set of objects that it owns, in the form of a map from objects to Boolean, so o->owns[x] is true iff o owns x. It is a system invariant that these sets form a partitioning of the set of objects that are not owned by any thread, and that open objects are owned only by threads. We call a thread-local object *mutable* (in the context of the thread) if it is open; we call a thread local object *wrapped* (in the context of the thread) if it is closed; a thread can only modify the non-volatile (aka sequential) fields of an object when it is mutable.

As an example, consider the following definition of a spinlock in annotated C code:

```
typedef struct _LOCK {
  volatile int locked;
  spec( obj_t prot_obj; )
  invariant( 0 == locked ==> owns[prot_obj])
} LOCK;
```

The type **obj_t** is a built-in type of typed pointers; a value of this type is a tuple consisting of an address and a type for which the address is suitably aligned. The macro **spec** indicates that its argument is ghost code included for purpose of proof but not part of the implementation. Thus, prot_obj is a ghost field of the lock. The lock uses a volatile integer field locked to keep track of its locked status (to be modified via atomic test-and-set operations). Finally, it has a (one-state) invariant that says that whenever the lock is available, the object protected by the lock is owned by the lock itself. Thus, the invariant above holds in the post-state of any transition for which the lock is closed in either the pre-state or the post-state. Note that because prot_obj is nonvolatile, the protected object cannot be changed as long as the lock remains closed (which, in case of a lock, is its normal state until it is eventually destroyed or used to protect a different object.)

Note that by having a generic object protected by the lock, we can have a single lock implementation that works with any protected object (which can itself own any collection of protected data, governed by any invariant).

2.3 Wrapping and Unwrapping

Unwrapping an object is the process of opening a wrapped object and transferring ownership of its owned objects to the thread. Conversely, wrapping an object requires that the object is mutable, and has the effect of transferring ownership of some specified wrapped objects to the object and closing the object. Thus, a thread typically modifies a shared object by taking ownership of the object (from another object), unwrapping it, modifying it, wrapping it, and putting it back somewhere (usually in its original place).

As an example, the following implementation of the lock initialization function shows the form of our function specifications and the use of wrap:

```
void InitializeLock(LOCK *l spec(obj_t obj))
  requires(mutable(l))
  requires(wrapped(obj))
  ensures(wrapped(l))
  ensures(l->locked == 0 && l->prot_obj == obj)
  writes(l, obj)
{
  l->locked = 0;
  spec( l->prot_obj = obj; )
  spec( wrap(l, { obj }); )
  return l;
}
```

In addition to the LOCK being initialized, the function takes a ghost parameter giving the protected object. This object must be wrapped, which means that it is owned by the current thread and is closed (hence the invariant on the protected data holds). As a postcondition, the function guarantees that the lock is wrapped, available, and bound to the protected object which (by the LOCK's invariant and the aforementioned invariants) guarantees that the protected object stays closed until the LOCK is acquired.

2.4 Claims

Threads can only meaningfully interact with shared state if they know something about that state. Since object invariants hold only when an object is closed, useful shared state information can be obtained only from objects that are known to be closed. A thread can attempt to acquire a LOCK only if it knows that the LOCK is closed. A thread can keep objects that it owns closed, so the issue arises only for objects that are not wrapped. Of course, this is the typical case when dealing with synchronization objects like locks whose whole purpose is to control the mutually exclusive ownership of other data.

One solution would be to use the object invariant to prevent the object from ever being opened, but such an object could never be destroyed, which makes this approach unsuitable in the usual case where such objects are dynamically allocated and deallocated.

Our solution is the introduction of *claims* to objects. Such a claim is a ghost object that stores a reference to its claimed object and has the invariant that the claimed object is closed. To ensure this, any object has an implicit ghost field that keeps track of the currently outstanding claims on that object; it also has a 2-state invariant that prevents it from going from the closed to the open state when this claims set in non-empty. As the final ingredient, the claim's invariant asserts membership in its claimed object's claims set.

In its simplest form, a claim can be thought of as handle to an object that guarantees the claimed object stays closed as long as the handle stays closed. (More elaborate claims can be constructed by referencing more than one object at once or putting additional invariants on the claim.) By creating multiple handles on a LOCK, multiple threads can now share the LOCK to gain exclusive access to its protected object:

[5] Note that in most cases this matches the meaning of a volatile field in C, i.e., a field that can be modified without the compiler noticing.

```
void Acquire(LOCK *l spec(claim_t c))
  requires(wrapped(c) && c->claimed_obj == l)
  ensures(wrapped(l->prot_obj))
  ensures(! old(owns(me)[l->prot_obj]))

void Release(LOCK *l spec(claim_t c))
  requires(wrapped(c) && c->claimed_obj == l)
  requires(wrapped(l->prot_obj))
  writes(l->prot_obj)
```

Note how `Acquire` requires, as a ghost parameter, a claim that guarantees that the lock is closed. It ensures that the protected object is wrapped, which implies that it is owned by the current thread (that has just completed the call to `Acquire`) and closed, thus its invariant holds. Also, by guaranteeing that the protected object has not been in the owns set of the current thread (represented by `owns(me)`), the current thread is free to change the object without interference with the rest of its ownership domain.

Dually, `Release` requires a claim on the lock and the protected object to be wrapped and ensures that the protected object is no longer a member of the ownership domain of the current thread. Note that `Release` does not ensure that the lock is unlocked. Indeed, depending on the scheduling, another thread could have acquired the lock after the current thread's release but before returning from the call to Release.

This example uses claims only in their simplest form. In addition to guaranteeing that its claimed objects are closed, a claim can state properties of the system state. The admissibility check for such properties amounts to checking that it is true at the time the claim is closed, and is preserved by changes to other objects (typically making use of the fact that the referenced objects remain closed).

For example, when a thread reads (or writes and retains information about) a shared variable, it normally constructs a claim that captures whatever information it needs to retain from this access. Thereafter, it doesn't have to recheck this information (even if it writes to shared state) until it choses to destroy the claim. By making claims explicit objects, they can be put inside of other objects, allowing arbitrary interlinking of knowledge about the system state[6].

The implementation of claims inside the system is described in Section 6.

2.5 Volatile Fields and Atomic Updates

A thread can use a claim to guarantee that an object is closed. But so far, we have only allowed objects to be modified when they are open, and a shared object cannot be opened (because there may be outstanding claims on it), so a claim would allow us only to read the object, not to modify it. To overcome this, we introduce atomic updates, which allow a thread to modify volatile fields of objects. The update requires that the modified objects are closed (e.g., by owning the objects or having claims on them), and also requires that the update preserves the 2-state invariants of the modified objects.

This is illustrated by the following implementation of `Acquire`. It spins, attempting in each operation to change the `locked` field of the lock from `0` to `1` in an atomic test-and-set operation[7]. When it finds that the bit has been successfully changed (i.e., if the bit was `0` before the operation, as indicated by the return value of the test-and-set operation), ownership of the protected object is moved from the lock to the current thread (by the `remove_from` operation) and the loop is terminated.

$$\begin{array}{llcl}
\text{fields} & f \in \mathbb{F} & ::= & \texttt{field} \mid f' \\
\text{variables} & x \in \mathbb{V} & ::= & \texttt{var} \mid x' \\
\text{expressions} & e & ::= & x \mid e_0 \cup e_1 \mid e_0 \setminus e_1 \mid \{e_0, \ldots, e_n\} \\
\text{reads} & r & ::= & x := e \mid x := e {\rightarrow} f \\
\text{updates} & u & ::= & e_0 {\rightarrow} f := e_1 \\
\text{actions} & a & ::= & r \mid \textbf{alter } e \textbf{ with } \overline{u} \\
\text{steps} & s & ::= & a \mid \textbf{atomic } \overline{a} \\
 & \overline{u} & ::= & u \,;\, \overline{u} \mid \epsilon \\
 & \overline{a} & ::= & a \,;\, \overline{a} \mid \epsilon \\
 & \overline{s} & ::= & s \,;\, \overline{s} \mid \epsilon
\end{array}$$

Figure 1. The language

```
int acquired = 0;
do {
  atomic (l) {
    if (!InterlockedBitTestAndSet(&l->locked, 0)) {
      acquired = 1;
      spec( remove_from(l->prot_obj, l); )
    }
  }
} while (!acquired);
```

The argument to the atomic block (`l` in the example) gives the set of objects whose volatile fields can be modified in the block. These objects are required to be closed, and their invariants are checked across the atomic transition.

2.6 Reduction

Within each thread, execution is broken down into a sequence of *actions*, each of which preserves all object invariants. However, since reasoning in the context of a thread may involve formulas that mention the states of objects not owned by the thread, reasoning is made easier by minimizing the number of places at which we have to consider the possibility of another thread changing the state. When reasoning within a thread, we consider the possibility of interruption only when the thread is about to communicate with other threads by reading or writing data outside its ownership domain. We show this reasoning is sound, even if the program is run under a scheduler that can interrupt the thread at any time.

3. Language

To describe more precisely the proof obligations generated in verification, we describe our method for a simplified language, abstract syntax of which is given in Figure 1. The language is stripped of control flow structures; these can easily be added. We have also omitted function calls: handling of function calls in the implementation is described in Section 7.

A program in the language is defined by specifying the sequence of steps ($\overline{s}$) for each thread. Each step is either an action or a sequence of actions grouped in an **atomic** block. Each action is either a read or a sequence of updates[8] grouped using an **alter** block (which additionally specifies the set of updated objects); each action is supposed to preserve all object invariants when executed atomically.

Expressions (e) include variable references and set operations. For brevity we skip base types and arithmetic.

[6] A more interesting use of claims in locks is assuring that when a lock is destroyed, it actually owns the protected object (i.e., the lock is free). One way to achieve this guarantee is to strengthen the lock invariant so that the lock always owns either the protected object (when free) or a claim that the lock is closed (when locked).

[7] Normally the compiler would not allow function calls (like the one to `InterlockedBitTestAndSet`) in an atomic block, but the CPU primitives are treated specially, essentially by inlining them.

[8] There would be no harm to allowing **alter** blocks to contain reads also, but in the absence of function calls these can be easily moved outside the scope of the **alter** block.

$$\begin{array}{lcl} [x]_{\mathcal{E}} & = & \mathcal{E}(x) \\ [e_0 \cup e_1]_{\mathcal{E}} & = & S^{-1}(S([e_0]_{\mathcal{E}}) \cup S([e_1]_{\mathcal{E}})) \\ [e_0 \setminus e_1]_{\mathcal{E}} & = & S^{-1}(S([e_0]_{\mathcal{E}}) \setminus S([e_1]_{\mathcal{E}})) \\ [\{e_0, \ldots, e_n\}]_{\mathcal{E}} & = & S^{-1}(\{[e_0]_{\mathcal{E}}, \ldots, [e_n]_{\mathcal{E}}\}) \end{array}$$

$$\frac{\mathcal{E}, W \vdash \langle \sigma[[e_0]_{\mathcal{E}}, f := [e_1]_{\mathcal{E}}], \overline{u} \rangle \hookrightarrow_t^* \sigma' \quad [e_0]_{\mathcal{E}} \in W}{\mathcal{E}, W \vdash \langle \sigma, e_0 \rightarrow f \mathrel{\mathbf{:=}} e_1 \,\mathbf{;}\, \overline{u} \rangle \hookrightarrow_t^* \sigma'} \qquad \frac{}{\mathcal{E}, W \vdash \langle \sigma, \epsilon \rangle \hookrightarrow_t^* \sigma}$$

$$\frac{\mathit{alter}_1(t, A, [e]_{\mathcal{E}}, \sigma) \quad \mathcal{E}, [e]_{\mathcal{E}} \vdash \langle \sigma, \overline{u} \rangle \hookrightarrow_t^* \sigma' \quad \mathit{alter}_2(t, A, [e]_{\mathcal{E}}, \sigma, \sigma') \quad \mathit{good}(\sigma, \sigma')}{\langle \langle \sigma, \mathcal{E} \rangle, \textbf{alter } e \textbf{ with } \overline{u} \rangle \rhd_{t,A} \langle \sigma', \mathcal{E} \rangle}$$

$$\frac{}{\langle \langle \sigma, \mathcal{E} \rangle, x \mathrel{\mathbf{:=}} e \rangle \rhd_{t,A} \langle \sigma, \mathcal{E}[x := [e]_{\mathcal{E}}] \rangle} \qquad \frac{\mathit{readable}(t, A, \sigma, \langle [e]_{\mathcal{E}}, f \rangle)}{\langle \langle \sigma, \mathcal{E} \rangle, x \mathrel{\mathbf{:=}} e \rightarrow f \rangle \rhd_{t,A} \langle \sigma, \mathcal{E}[x := \sigma([e]_{\mathcal{E}}, f)] \rangle}$$

$$\frac{\langle \langle \sigma, \mathcal{E} \rangle, a \rangle \rhd_{t,A} \langle \sigma', \mathcal{E}' \rangle \quad \langle \langle \sigma', \mathcal{E}' \rangle, \overline{a} \rangle \rhd_{t,A}^* \langle \sigma'', \mathcal{E}'' \rangle}{\langle \langle \sigma, \mathcal{E} \rangle, a \,\mathbf{;}\, \overline{a} \rangle \rhd_{t,A}^* \langle \sigma'', \mathcal{E}'' \rangle} \qquad \frac{}{\langle \langle \sigma, \mathcal{E} \rangle, \epsilon \rangle \rhd_{t,A}^* \langle \sigma, \mathcal{E} \rangle}$$

$$\frac{T(t) = \langle \mathcal{E}, \textbf{atomic } \overline{a} \,\mathbf{;}\, \overline{s} \rangle \quad \langle \langle \sigma, \mathcal{E} \rangle, \overline{a} \rangle \rhd_{t,1}^* \langle \sigma', \mathcal{E}' \rangle}{\langle \sigma, T \rangle \blacktriangleright_t \langle \sigma', T[t := \langle \mathcal{E}', \overline{s} \rangle] \rangle} \qquad \frac{T(t) = \langle \mathcal{E}, a \,\mathbf{;}\, \overline{s} \rangle \quad \langle \langle \sigma, \mathcal{E} \rangle, a \rangle \rhd_{t,0} \langle \sigma', \mathcal{E}' \rangle}{\langle \sigma, T \rangle \blacktriangleright_t \langle \sigma', T[t := \langle \mathcal{E}', \overline{s} \rangle] \rangle}$$

Figure 2. The semantics

3.1 The semantics

For simplicity of exposition, we use a single data type $\mathbb{P}$, an infinite enumerable set of *pointers*. Since for some purposes we need to interpret pointers as sets, assume a bijection S from pointers to finite sets of pointers.

The semantics of the language is defined in terms of transitions between configurations $\langle \sigma, T \rangle$, where each configuration consists of a a heap $\sigma : \mathbb{P} \times \mathbb{F} \rightarrow \mathbb{P}$ and a function T that maps each thread to a local environment and a continuation, i.e. $T(t) = \langle \mathcal{E}, \overline{s} \rangle$ where $\mathcal{E} : \mathbb{V} \rightarrow \mathbb{P}$ is thread t's local environment and $\overline{s}$ are the remaining steps of thread t.

The thread-specific transition relation $\langle \sigma, T \rangle \blacktriangleright_t \langle \sigma', T' \rangle$ defined in Figure 2 describes the effect of running a single step of thread t on state $\langle \sigma, T \rangle$. It selects the environment and the next action of the thread t from T and executes is according to either $\rhd_{t,0}$ or $\rhd_{t,1}$ depending on whether execution is inside or outside of an atomic block. The $0/1$ flag is used in side conditions checked when reading or writing the state.

The conditions $\mathit{alter}_1(\ldots)$, $\mathit{alter}_2(\ldots)$, $\mathit{good}(\ldots)$ and $\mathit{readable}(\ldots)$ are used to enforce our methodology. If an execution fails to satisfy either of those conditions, it is said to *go wrong*, that is reach a special configuration $\bot$. We generate verification conditions stating that the program never goes wrong, but we do not expect the real machine to check them at runtime (see Section 5 for details).

Definitions of those side conditions require introduction of a few functions constituting our system invariants.

Predicate $\mathit{thread}(p)$ is true iff given pointer represents a thread. Predicate $\mathit{volatile}(p, f)$ is true iff field f of p is to be considered volatile, i.e. is allowed to change without the object p being opened (see definition of $\mathit{non\_vol}(\ldots)$ for details). Both are state-independent and in the implementation are derived from type definitions. Whenever we use a variable t (possibly with indices) we implicitly assume $\mathit{thread}(t)$.

The predicate $\mathit{closed}(\sigma, o)$, stating that o is closed in state σ, is defined as $\sigma(o, \texttt{closed}) = 1$, where 1 is treated as a distinguished element of $\mathbb{P}$. Similarly we define the owns set function $\mathit{owns}(\sigma, o)$ as $S(\sigma(o, \texttt{owns}))$.

The ownership domain of p in state σ, $\mathit{domain}(\sigma, p)$, is the minimal solution to

$$\mathit{domain}(\sigma, p) = \{p\} \cup$$

$$(\cup q : \neg \mathit{volatile}(p, \texttt{owns}) \ \wedge \ q \in \mathit{owns}(\sigma, p) : \mathit{domain}(\sigma, q))$$

The partition of thread t in σ (written $\mathit{partition}(\sigma, t)$) is defined as the set of those $\langle p, f \rangle$ such that

$$p \in \mathit{domain}(\sigma, t) \ \wedge \ (\neg \mathit{volatile}(p, f) \ \vee \ \neg \mathit{closed}(\sigma, p))$$

The shared portion of state σ is everything outside any partition:

$$\mathit{shared}(\sigma) = \{\langle p, f \rangle \mid \neg \exists t. \ \langle p, f \rangle \in \mathit{partition}(\sigma, t)\}$$

The condition $\mathit{readable}(t, A, \sigma, l)$ under which a thread is allowed to read data is

$$l \in \mathit{partition}(\sigma, t) \ \vee \ (A \ \wedge \ l \in \mathit{shared}(\sigma))$$

In other words, a thread is allowed to read its partition, and can also read shared data when inside an atomic block.

The conditions for writing are three-fold. First we need to make sure the current thread either owns the data it is going to write or it is running atomically and the object written is closed, i.e. $\mathit{alter}_1(t, A, W, \sigma)$ is:

$$\forall p \in W. \ (A \ \wedge \ \mathit{closed}(\sigma, p)) \ \vee \ p \in \mathit{owns}(\sigma, t)$$

Second, the thread is not allowed to open or close objects outside of its owns set nor to change its partition (by opening or closing an object with volatile fields) while outside of an atomic block, i.e. $\mathit{alter}_2(t, A, W, \sigma, \sigma')$ is:

$$\begin{array}{l} \forall p \in W. \ \mathit{closed}(\sigma, p) \neq \mathit{closed}(\sigma', p) \Rightarrow \\ \quad p \in \mathit{owns}(\sigma, t) \ \wedge \ (A \ \vee \ \forall f. \ \neg \mathit{volatile}(p, f)) \end{array}$$

Finally we want to ensure that the update preserves higher-level system invariants as well as object invariants, that is $\mathit{good}(\sigma, \sigma')$ is defined as:

$$\mathit{closed}^*(\sigma') \ \wedge \ \mathit{non\_vol}(\sigma, \sigma', S) \ \wedge \ \mathit{inv}_{\neq}(\sigma, \sigma')$$

The predicate $\mathit{closed}^*(\sigma)$, stating the interactions between the `closed` and `owns` fields is defined as conjunction of:

1. $\forall p, q. \ \mathit{closed}(\sigma, p) \ \wedge \ q \in \mathit{owns}(\sigma, p) \Rightarrow \mathit{closed}(\sigma, q)$
   (if you are closed then everything you own is closed)

2. $\forall p.\ \neg thread(p)\ \wedge\ \neg closed(\sigma, p) \Rightarrow owns(\sigma, p) = \emptyset$ (ordinary open objects cannot own anything)
3. $\forall p.\ thread(p) \Rightarrow \neg closed(\sigma, p)\ \wedge\ p \in owns(\sigma, p)$ (threads cannot be closed and own themselves)
4. $\forall p, q.\ owns(\sigma, p) \cap owns(\sigma, q) \neq \emptyset \Rightarrow p = q$ (owns sets are disjoint)
5. $\forall p.\ \exists q.\ p \in owns(\sigma, q)$ (everyone is owned)

The last two condition amount to saying that there exists an $owner(\sigma, o)$ function giving the only object in owns set of which o is contained.

The predicate $non\_vol(\sigma, \sigma')$ says that the non-volatile fields of closed objects did not change:

$$\forall p, f.\ closed(\sigma, p)\ \wedge\ closed(\sigma', p)\ \wedge\ \neg volatile(p, f) \Rightarrow \sigma(p, f) = \sigma'(p, f)$$

Finally the predicate $inv_{\neq}(\sigma, \sigma')$ says that the two-state invariants of objects changed by the alter block has been preserved:

$$\forall p.\ closed(\sigma, p)\ \vee\ closed(\sigma', p) \Rightarrow \sigma =_p \sigma'\ \vee\ inv(\sigma, \sigma', p)$$

where $\sigma =_p \sigma'$ is defined as $(\forall f.\ \sigma(p, f) = \sigma'(p, f))$.

The predicate $inv(\sigma, \sigma', p)$ is the two-state invariant for pointer p. In our implementation, this is calculated from the type declaration of p.

3.2 Admissibility

DEFINITION 1. *We say the invariant of object* p *is admissible iff for any σ and σ' if:*

1. $\sigma =_p \sigma'$;
2. $non\_vol(\sigma, \sigma')$;
3. $closed(\sigma', p)$;
4. $\forall o.\ closed(\sigma, o) \Rightarrow inv(\sigma, \sigma, o)$;
5. $inv_{\neq}(\sigma, \sigma')$; *and*
6. $closed^*(\sigma)\ \wedge\ closed^*(\sigma')$,

then:

1. $closed(\sigma, p) \Rightarrow inv(\sigma, \sigma', p)$ *(stability); and*
2. $inv(\sigma, \sigma', p) \Rightarrow inv(\sigma', \sigma', p)$ *(stuttering).*

Our system requires the user to prove (from type and invariant declarations alone) that all invariants declared on types are admissible. Henceforth we assume that all invariants are admissible.

Let $sinv_2(\sigma, \sigma') = \forall p.\ closed(\sigma, p)\ \vee\ closed(\sigma', p) \Rightarrow inv(\sigma, \sigma', p)$, i.e., the invariants of all closed objects hold between σ and σ'. Let $sinv_1(\sigma) = closed^*(\sigma)\ \wedge\ inv(\sigma, \sigma)$ be the invariant that every state of execution will fulfill.

If all invariants are admissible, we only need to check invariants of object that were updated:

LEMMA 1. *If $sinv_1(\sigma)$ and $good(\sigma, \sigma')$ then $sinv_2(\sigma, \sigma')$ and $sinv_1(\sigma')$.*

Proof. Trivial □

4. Wrap and unwrap

Because checking of $closed^*(...)$ can be difficult in general, we introduce the wrap and unwrap operations that, under certain conditions, are guaranteed to maintain $closed^*$ and present them to the user as the only way of opening and closing objects. They are defined as:

```
wrap $e_0, e_1$ =
  tmp := me→owns
  alter {$e_0$, me} with
    $e_0$→owns := $e_1$
    me→owns := tmp \ $e_1$
    $e_0$→closed := 1
```

```
unwrap $e_0$ =
  tmp$_0$ := $e_0$→owns
  tmp$_1$ := me→owns
  alter {$e_0$, me} with
    $e_0$→owns := {}
    me→owns := tmp$_0$ ∪ tmp$_1$
    $e_0$→closed := 0
```

where the variable `me` refers to the current thread.

The wrap operation requires e_0 to be an open object, owned by `me` and all objects in e_1 to be closed and owned by `me`. Additionally, if there are any volatile fields in e_0, it can only be performed inside of an atomic block. Under these conditions, wrapping satisfies the $alter_1(...)$ and $alter_2(...)$ conditions. Additionally, it satisfies the $closed^*(...)\ \wedge\ non\_vol(...)$ part of $good(...)$. Checking of the invariant of the object being wrapped remains as a proof obligation.

The unwrap operation requires e_0 to be closed and owned by `me`. As in the case of wrap, having volatile fields in e_0 adds the requirement that the unwrap needs to be performed inside of an atomic block. Again, if the invariant of e_0 allows opening it, unwrapping satisfies the side conditions on **alter**.

Additionally we introduce two operations to move an object e_0 in and out of a volatile owns set of another object e_1. The object e_0 is moved to/from the owns set of the current thread:

```
put $e_0$ in $e_1$ =
  tmp$_0$ := me→owns
  tmp$_1$ := $e_1$→owns
  alter {$e_1$, me} with
    me→owns := tmp$_0$ \ {$e_0$}
    $e_1$→owns := tmp$_1$ ∪ {$e_0$}
```

```
remove $e_0$ from $e_1$ =
  tmp$_0$ := me→owns
  tmp$_1$ := $e_1$→owns
  alter {$e_1$, me} with
    me→owns := tmp$_0$ ∪ {$e_0$}
    $e_1$→owns := tmp$_1$ \ {$e_0$}
```

Both need to be executed inside of an atomic block and require e_0 and e_1 to be closed. Additionally e_1 is required to have a volatile `owns` field. The **put** operation additionally requires the object e_1 to be in the owns set of the current thread, while the **remove** requires it to be in the owns set of e_0. Again, checking that the invariant of e_0 allows adding/removing a child is a proof obligation.

5. Reduction

On a real machine, a program executes under a fine-grained scheduler that allows thread switching at any time. However, in reasoning about a program, we assume a coarse-grained scheduler that switches back to a previously scheduled thread only when it is about to execute an atomic block. The reduction theorem below justifies this pretense (but see Section 9 for further discussion on this point).

Let the relation $\tilde{\blacktriangleright}_t$ be defined as $\blacktriangleright_t$ in Figure 2, but with conditions $readable(...)$, $alter_1(...)$, $alter_2(...)$ and $good(...)$ all replaced by true. In other words it refers to the real machine, not executing any runtime checks.

We say that a configuration $c' = \langle \sigma', T' \rangle$ is reachable from $c = \langle \sigma, T \rangle$, written $c \blacktriangleright^* c'$ iff there exists a sequence:

$$c = c_0 \blacktriangleright_{t_0} c_1 \blacktriangleright_{t_1} \ldots \blacktriangleright_{t_{n-1}} c_n = c'$$

DEFINITION 2. *A sequence of transitions:*

$$\langle \sigma_0, T_0 \rangle \blacktriangleright_{t_0} \langle \sigma_1, T_1 \rangle \blacktriangleright_{t_1} \ldots \blacktriangleright_{t_{n-1}} \langle \sigma_n, T_n \rangle$$

*is a coarse schedule if, for all i where $0 \leq i < n$, either $t_i = t_{i+1}$, $T_{i+1}(t_{i+1})$ is of the form $\langle$**atomic** $\ldots; \ldots, \ldots\rangle$, or $t_j \neq t_{i+1}$ for all $0 \leq j < i$.*

The following theorem is proved in the appendix. It says that if a program can go wrong under an arbitrary schedule, it can go wrong under a coarse schedule. Thus, in proving that a program doesn't go wrong, we can assume coarse scheduling.

THEOREM 1. *(soundness of coarse scheduling)*
For any sequence of transitions:

$$\langle \sigma_0, T_0 \rangle = c_0 \ \tilde{\blacktriangleright}_{t_0}\ c_1 \ \tilde{\blacktriangleright}_{t_1}\ \ldots\ \tilde{\blacktriangleright}_{t_{n-1}}\ c_n$$

if $sinv_1(\sigma_0)$ *and* $\neg(c_0 \blacktriangleright^* \perp)$ *then there exists a coarse schedule*

$$c_0' \blacktriangleright_{t_0'} c_1' \blacktriangleright_{t_1'} \ldots \blacktriangleright_{t_{n-1}'} c_n'$$

such that $c_0 = c_0'$ *and* $c_n = c_n'$.

6. Claims

A *type definition* has the form **type** $t = \{\overline{F}\}$ where:

$$\overline{F} \ ::= \ \textbf{volatile}\ f\,\mathbf{;}\,\overline{F} \mid f\,\mathbf{;}\,\overline{F} \mid \epsilon$$

It is syntactic sugar for introducing a predicate t defined on pointers and constraining the $volatile(...)$ function so for **type** $t = \{\ldots\mathbf{;}\,\textbf{volatile}\ f\,\mathbf{;}\ldots\}$ we have $t(p) \Rightarrow volatile(p, f)$ and for **type** $t = \{\ldots\mathbf{;}\, f\,\mathbf{;}\ldots\}$ we have $t(p) \Rightarrow \neg volatile(p, f)$.

Consider the following type definitions:

$$\textbf{type}\ Data = \{\textbf{volatile}\ \mathtt{H}\} \quad \textbf{type}\ Handle = \{\mathtt{P}\}$$

The idea is that if you own a h such that $Handle(h)$ you can rest assured that the $\sigma(h, \mathtt{P})$ will stay closed (where presumably $Data(\sigma(h, \mathtt{P}))$). To make sure $Data$ knows about it we include a set of active handles in it. The $inv(\sigma_0, \sigma, h)$ will imply:

$$Handle(h) \Rightarrow h \in \sigma(\sigma(h, \mathtt{P}), \mathtt{H}) \ \wedge \ closed(\sigma, \sigma(h, \mathtt{P}))$$

To make this invariant admissible we need to also restrict changes to the date, i.e. the $inv(\sigma_0, \sigma, d)$ should imply:

$$\begin{aligned} &Data(d) \Rightarrow \\ &\qquad (\forall h.\ closed(\sigma, h) \Rightarrow (h \in \sigma(d, \mathtt{H}) \Leftrightarrow \sigma(h, \mathtt{P}) = d)) \\ &\quad \wedge (closed(\sigma_0, d) \ \wedge \ \neg closed(\sigma, d) \Rightarrow \sigma(d, \mathtt{H}) = \emptyset) \\ &\quad \wedge (\forall o.\ \neg thread(o) \ \wedge \ p \in owns(\sigma, o) \Rightarrow \\ &\qquad\qquad \sigma_0(d, \mathtt{H}) = \sigma(d, \mathtt{H}) \ \vee \ inv(\sigma_0, \sigma, o)) \end{aligned}$$

Note that there is a possible circularity problem with a dependency of $inv(\sigma_0, \sigma, d)$ on $inv(\sigma_0, \sigma, o)$. To guarantee consistency, such terms can occur only with positive polarity in object invariants.

The owner of the data can control the handle set, for example given: **type** $Ctrl = \{\mathtt{D};\ \mathtt{H}\}$ where $inv(\sigma_0, \sigma, c)$ implies:

$$Ctrl(c) \Rightarrow \sigma(c, \mathtt{D}) \in owns(\sigma, c) \ \wedge \ \sigma(\sigma(c, \mathtt{D}), \mathtt{H}) = \sigma(c, \mathtt{H})$$

That is the controller includes a non-volatile copy of the handle set. Because the invariant of the $Data$ says that whenever changing its handle set it will check with the invariant of the owner, the invariant of $Ctrl$ is admissible.

For example to create a handle h, given a wrapped controller c we would do:

```
atomic
  d := $c \to \mathtt{D}$
  $h \to \mathtt{P} := d$
  unwrap $c$
  alter $\{d\}$ with
    tmp := $d \to H$
    $d \to H :=$ tmp $\cup \{h\}$
    $c \to H :=$ tmp $\cup \{h\}$
  wrap $h, \{\}$
  wrap $c, \{d\}$
```

The inclusion of the handle set in the $Ctrl$ allows for restriction on creation of new handles. This is used to implement concurrency primitives like reader-writer locks. Whenever a new reader lock is acquired, a new handle is created and given out to the caller. When the reader lock is released, the caller needs to give back the handle, which we open up and remove from the handle set. This way the volatile integer counting the number of reader locks is tied to the cardinality of the handle set, and thus if its zero the $Data$ can be open (and for example given out to a caller acquiring a writer lock).

A claim builds on top of a handle. It is an object owning one or more handles. When checking admissibility of invariant of the claim we can rely on the fact that the objects to which we hold handles are closed, and thus respect their invariants. Because the claim is first closed in a particular state, the invariant of the claim can depend on some properties of that state. One example would be that a claim on an object with a field that can never decrease could have an invariant guaranteeing the value to be at least the value at the time when the claim was taken.

Our implementation includes claims as built-ins to allow for creation of claims in the running code of a function. This overcomes some practical problems, where parts of the local state would need to be copied to fields of claim, so they can be mentioned in the invariants. There is however no theoretical reason to do that — the methodology primitives are strong enough to implement claims.

7. Framing

Our implementation allows functions with pre-, post-conditions and writes clauses. The interpretation of the pre- and post-conditions is standard. The function is allowed to write to the sequential ownership domain of objects listed in its writes clause, as defined before the call. This is enforced by maintaining a ghost variable with the set of currently writable objects. The set is initialized to the writes clause, and when unwrapping o, the owns set of o is added to the writes clause. Upon writing to p the function is required to check that p is in the writes set and is owned by the current thread. This way the wrap operation does not need to shrink the writes set.

The writes set is however not consulted when performing writes to the shared state in the atomic block – these are always allowed (as permitted by invariants). On the other hand, if an atomic operation causes objects to be moved into owns set of the current thread, such objects are also added to the writes set. This is a generalization of object allocation (moving it from the ownership domain of the memory allocator to the ownership domain of the current thread).

8. Verification Conditions

We generate verification conditions (VCs) for each method. They state that if we consider the configuration $\langle \sigma, T \rangle$ where $T(t) = \langle \mathcal{E}, \overline{s} \rangle$ where $\overline{s}$ is the body of the function and σ and $\mathcal{E}$ satisfy the preconditions of the function, then the state $\perp$ will not be reached by the transition of thread t, executed under a coarse scheduler. The VCs explicitly check invariants of changed objects, as well as $readable(...)$ and $alter_1(...)$ but the $closed^*(...)$ predicate is enforced by the wrap/unwrap protocol. The VCs assume $sinv_1(...)$ to hold before every transition.

The encoding of framing is based on (Barnett et al. 2004): because the definition of the ownership domain uses the reachability relation, which is expressible in first order logic, we over approximate the set of possibly written objects by saying that, in addition to objects listed in the writes clause, every object not owned by the current thread could have changed. Additionally any volatile field of any object could have changed. Such a weakening is also used to simulate actions of other threads at the entry of the atomic block (with an empty writes clause).

However, as an extension, we allow user to manually annotate code to tell the prover that before this particular call the object o was in the ownership domain of object o'. Such an assertion is checked, but if the check succeeds we assume that the object o cannot change if the object o' does not change. Thus, if the object o' is owned by the thread, and not listed in the writes clause, we will also know that o did not change.

Such an assumption is sound, because we know the real semantics of the writes clause, namely writing only the ownership domain of the objects listed.

9. Other Issues

Here we make mention of some methodological issues that go beyond the scope of the paper, but are important for soundness or methodological reasons.

9.1 Ghost Data versus Real Data

In the presentation, we have treated real data/code and ghost data/-code uniformly. There are several important differences between them.

In C, pointers can be injectively cast to sufficiently long unsigned integer types. This puts concrete limits on the physical address space, so ghost data has to be located at "large" addresses outside of this range. Similarly, ghost fields of objects cannot be located within the object like real fields within C, so ghost fields have to be formalized as maps from pointers of appropriate type to addresses of the corresponding ghost fields.

In order to guarantee that the real code (where all ghost data is erased) simulates the verified code, all ghost code has to be total and terminating[9]. Historically, methodologies achieve this by not allowing ghost code to effect control flow. However, for pragmatic reasons, we want to manipulate ghost state with functions. Therefore, we don't allow recursion in ghost functions, or iterations within ghost code, even though less draconian approaches are possible. This would be more problematic were it not for the fact that we can do large (first-order) updates to ghost state in a single step[10].

9.2 Compiler and Scheduler Issues

We have assumed in the operational model that programs are executed according to the program text, with thread switching only at action boundaries. In a realsic implementation, thread switching can occur anywhere outside of an atomic block. Moreover, C compilers are allowed to move around (or even modify) accesses to nonvolatile data in between volatile accesses. While the details of the allowed optimizations go beyond the scope of this paper, we have proved that the reasoning described here remains sound even under these more realistic schedulers and compilers.

10. Evaluation

The proposed verification methodology has been implemented in an automated, sound C verifier being used to verify the functional correctness of the Hypervisor (the virtualization kernel of Microsoft's Hyper-V product(Microsoft 2008)). This verifier translates annotated C code into BoogiePL (DeLine and Leino 2005), an intermediate language for verification. The verification condition generator Boogie (Barnett et al. 2006) takes BoogiePL as input, and feeds the generated verification conditions into the Z3 (de Moura and Bjrner 2008) SMT solver.

The methodology as described in this paper has been developed and deployed over the last year. So far we have verified the functional correctness of Hyper-V's sequential circular lists, spin locks, rundowns, lockfree lists, reader-writers, and some algorithms using locks like Hyper-V's internal malloc. In addition we are in the process of finalizing the definition of all externally visible module invariants and we have checked their admissibility. We have also formalized the complex ownership and claim structure that exists between threads and processes and verify process creation and the adding and removing threads from processes. Verification has typically been in the range to up to 10 seconds per method.

The methodology has been developed as a consequence of an earlier unsuccessful attempt to use dynamic frames (Kassios 2006). Using this approach, we verified the C simulation of Windows based Smart Card a partial functional correctness of a "baby" hypervisor, several sequential parts of the Microsoft Hypervisor, including the memory safety of approx 4500 lines of the x86 assembly code by translating it into C and making the machine state explicit. However when we started to verify higher abstraction layers, where invariants were depending on object footprints, defined by invariants themselves, we quickly run into limitations of that approach. Furthermore, we could not find an elegant and effective way to extend the method for multi-threaded reasoning.

11. Related Work

Research on modular verification of concurrent programs has centered on one problem: reasoning about the interference between threads.

Owicki and Gries (Owicki and Gries 1976) introduced the concept of non-interference between the proofs of concurrent threads.

Jones (Jones 1981) introduced rely/guarantee pairs to reason compositionally about concurrent programs. Rely/guarantee pairs describe the state changes performed by the environment or by the program respectively. Unfortunately the specification of interference is global: it must be checked against every state update.

Several static checkers have been built to support Owicki/-Gries and Jones style reasoning. The Extended Static Checkers for Modula-3 and for Java (Detlefs et al. 1998) include support for the prevention of data races and deadlocks. For each field a programmer designate a lock, which protects it. But these two tools trade soundness for ease of use; for example, they do not take into consideration the effects of other threads between regions of exclusion.

Concurrent Spec# (Jacobs et al. 2007) is a sound verifier for concurrent, object-oriented C#. It imposes a hierarchical ownership discipline on object-oriented program. The ownership discipline guarantees the partition of the state into distinct object ownership domains and thread local data. This allows us to ignore concurrency largely. Locking and unlocking is modelled by ownership transfer operations between threads[11]. Interleaving of other threads is modelled by weakening what is known about the shared state. Unfortunately the expressive power of concurrent Spec# is restricted: Spec# supports only 1-state invariants, no volatiles and no abstractions via claims. Inspired by rely/guarantee reasoning, Spec# later introduced 2-state invariants, but only to verify the interaction between several sequential objects (Leino and Schulte 2007).

Concurrent Separation Logic (CSL) forbids interference except in critical regions (OHearn 2007b). CSL uses the footprint of an object's invariant as a foundation for partitioning the state [12]. However in CSL, object invariants cannot mention other objects. Some additional flexibility can be obtained using *read permissions* (Bornat et al. 2005), which allow objects to refer to shared read-only por-

[9] The positive side of this is that nondeterminism within ghost code is angelic rather than demonic. Our verifier does no take advantage of this fact, however.

[10] For example, to handle recursive data structures with first-order methods, we maintain the relevant reachability relations for the structure in ghost state. An update to a structure (e.g., adding a new node to the front of a linked list) requires updating the reachability relation for an unbounded number of elements in a single ghost step. Fortunately, the pairs added to the reachability relation can be given by a first-order relation (the new node reaches the first node and all nodes previously reached by the first node), so the whole reachability relation can be updated by a single first-order update.

[11] We model locks very similarly, with the advantage of not considering them as primitives, which allows for verifying their implementation.

[12] In the context of CSL, the objects are resources.

tions of the state; this allows invariants to cross object boundaries, but updating an object might require locking more objects than could be achieved using claims. Later CSL was improved upon by marrying it with rely/guarantee reasoning (Vafeiadis and Parkinson 2007). Rely/guarantee pairs, which are normally binary relations, are here expressed as set of of actions. Our admissibility check is similar to CSL obligations that invariants are stable under the rely/guarantee actions.

Recently shape analysis has also been used to verify concurrent data structures (Berdine et al. 2008). But unlike the earlier mentioned approaches concurrent shape analysis tries to find abstractions of the program state from the perspective of a particular thread. Also, shape analysis works on elements of abstract domains, and thus might loose precision.

Reduction theorems of the sort described in Section 5 are well-known from both concurrent programmiing (Cohen and Lamport 1998) and database concurrency control (Eswaran et al. 1976).

12. Conclusion

We have demonstrated that our approach is effective for proving concurrent programs on different abstraction levels. We have verified concurrent algorithms as diverse as spinlocks, lockfree lists, several algorithms that use lock-protected data, and we have started to verify Hyper-V's thread scheduler, which uses a highly concurrent queue. The postulated verification methodology is thread modular: ownership partitions the state space; sequential reasoning can be used for proving properties about thread local state; object invariants and claims controls the effect of interleavings on shared state. But while we can now verify those algorithms, the annotation overhead is high. In the future we will investigate how those annotations can be inferred automatically.

Acknowledgments

We should like to thank Andrey Shadrin, Dirk Leinenbach, Elena Petrova, Holger Blasum, Lieven Desmet, Mark A. Hillebrand, Sergey Tverdyshev, and Thomas Santen for discussions, adoption and feedback on this work. We would also like to thank Herman Venter for his help with the compiler framework.

References

Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008. ISBN 978-3-540-70543-7.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40 (1):259–270, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1047659.1040327.

Ernie Cohen and Leslie Lamport. Reduction in tla. In *In International Conference on Concurrency Theory*, pages 317–331. Springer-Verlag, 1998.

Leonardo de Moura and Nikolaj Bjrner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008. doi: http://dx.doi.org/10.1007/978-3-540-78800-3_24. URL `http://dx.doi.org/10.1007/978-3-540-78800-3\_24`.

Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360363.360369.

C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355620.361161.

Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electr. Notes Theor. Comput. Sci.*, 174 (9):23–47, 2007.

C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/69575.69577.

Cliff B. Jones. Development methods for computer programs including a notion of interference. Technical report, Oxford UNiversity, PhD thesis, 1981.

Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 032114306X.

K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2007. ISBN 978-3-540-71314-2.

Microsoft. Virtualization with hyper-v, 2008. http://www.microsoft.com/windowsserver2008/en/us/hyperv.aspx.

Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007a. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2006.12.035.

Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007b. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2006.12.035.

Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

Matthew Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1):297–302, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1190215.1190261.

Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. ISBN 978-3-540-74406-1.

A. Proof of the Reduction Theorem

In this section, we prove that it is safe to assume coarse scheduling when reasoning about programs (Theorem 1).

We will use the letter c to refer to configurations, that is pairs $\langle \sigma, T \rangle$.

FACT 1. *Given a pre-state and a thread to run the post-state is deterministic: If $c \mathrel{\tilde{\blacktriangleright}_t} c'$ and $c \mathrel{\tilde{\blacktriangleright}_t} c''$ then $c' = c''$ (and similarly for $\blacktriangleright_t$).*

DEFINITION 3. *The configurations $c = \langle \sigma, T \rangle$ and $c' = \langle \sigma', T' \rangle$ agree on partition of t, written $c \approx_t c'$ iff $\forall \langle p, f \rangle \in partition(\sigma, t).\ \sigma(p, f) = \sigma'(p, f)$ and $T(t) = T'(t)$.*

LEMMA 2. *Partition function is self-preserving: If $\langle \sigma, T \rangle \approx_t \langle \sigma', T' \rangle$ and $sinv_1(\sigma)$ then $partition(\sigma, t) = partition(\sigma', t)$.*

DEFINITION 4. *The configurations $c = \langle \sigma, T \rangle$ and $c' = \langle \sigma', T' \rangle$ agree outside of partition of t, written $\sigma \approx_{\bar{t}} \sigma'$ iff $\forall \langle p, f \rangle \notin partition(\sigma, t).\ \sigma(p, f) = \sigma'(p, f)$, $partition(\sigma, t) = partition(\sigma', t)$ and $\forall t' \neq t.\ T(t') = T'(t')$.*

LEMMA 3. *If $closed^*(\sigma)$ and $t_0 \neq t_1$ then $domain(\sigma, t_0) \cap domain(\sigma, t_1) = \emptyset$ (and thus also $partition(\sigma, t_0) \cap partition(\sigma, t_1) = \emptyset$).*

LEMMA 4. *A non-atomic transition of thread t depends only on its partition: If $c_0 \blacktriangleright_t c_1$, $c_0' \tilde{\blacktriangleright}_t c_1'$, $c_0 \approx_t c_0'$ and $T(t)$ does not start with* **atomic**, *then $c_1 \approx_t c_1'$, $c_0 \approx_{\bar{t}} c_1$ and $c_0' \approx_{\bar{t}} c_1'$.*

The following theorem states that for any sequence of transitions that does not involve running thread t, the sequence does not touch the partition of t (first two conditions) and moreover the sequence does not depend on changes of state within the partition of t (the last condition).

THEOREM 2. *For every t and every sequence of transitions:*

$$\langle \sigma_0, T_0 \rangle = c_0 \blacktriangleright_{t_0} c_1 \blacktriangleright_{t_1} \dots \blacktriangleright_{t_{n-1}} c_n$$

if $sinv_1(\sigma_0)$ and $t_i \neq t$ for $i = 0 \dots n-1$ then:

1. $c_0 \approx_t c_n$
2. for any c_0' if $c_0 \approx_{\bar{t}} c_0'$ then

$$c_0' \blacktriangleright_{t_0} c_1' \blacktriangleright_{t_1} \dots \blacktriangleright_{t_{n-1}} c_n'$$

and $c_n \approx_{\bar{t}} c_n'$.

Proof. By induction on the length of the sequence.

1. The only instruction that performs state modifications is **alter** e **with** $\overline{u}$. The updates $\overline{u}$ can only modify fields of objects listed in e. The $alter_1(...)$ conditions ensures these objects in the domain of the thread t_i, where $t_i \neq t$ and thus by Lemma 3 not in domain of t or are initially closed. Therefore the only interaction with the partition of t could be non-volatile fields of initially closed objects. However object outside the domain of t_i cannot be open ($alter_2(...)$), thus objects in domain of t will stay closed. Finally $non\_vol(...)$ ensures that all closed objects have their non-volatile fields unchanged.
2. The only operation that depends on σ is $x := e \rightarrow f$. It is however restricted to reading from outside of partition of t (from partitions of t_i and the shared part, which is not covered by any partition (including t's)).

□

We are now ready to prove the soundness of the assumption of coarse scheduling.

THEOREM 3. *(soundness of coarse scheduling)*
For any sequence of transitions:

$$\langle \sigma_0, T_0 \rangle = c_0 \tilde{\blacktriangleright}_{t_0} c_1 \tilde{\blacktriangleright}_{t_1} \dots \tilde{\blacktriangleright}_{t_{n-1}} c_n$$

if $sinv_1(\sigma_0)$ and $\neg(c_0 \blacktriangleright^* \bot)$ then there exists a coarse schedule

$$c_0' \blacktriangleright_{t_0'} c_1' \blacktriangleright_{t_1'} \dots \blacktriangleright_{t_{n-1}'} c_n'$$

such that $c_0 = c_0'$, and $c_n = c_n'$.

Proof. By induction on n. It trivially holds for $n \leq 3$. Now we want to prove it for a particular n assuming it holds for smaller ones. We use the induction hypothesis on

$$c_0 \tilde{\blacktriangleright}_{t_0} \dots \tilde{\blacktriangleright}_{t_{n-2}} c_{n-1}$$

getting a coarse schedule:

$$c_0' \blacktriangleright_{t_0'} \dots \blacktriangleright_{t_{n-2}'} c_{n-1}'$$

where $c_{n-1}' = c_{n-1}$.

Let $t = t_{n-1}$. We find the last operation of thread t in our coarse schedule, i.e. the biggest $k < n$ such that $t_k' = t$. If such k does not exists or $T_{n-1}(t)$ starts with **atomic** then the sequence:

$$c_0' \blacktriangleright_{t_0'} \dots \blacktriangleright_{t_{n-2}'} c_{n-1}' \blacktriangleright_t c_n$$

is a coarse schedule and we are done.

Otherwise we move the last operation of thread t_n directly after the t_k', i.e. consider the sequences:

$$\begin{array}{l} c_0' \blacktriangleright_{t_0'} \dots \blacktriangleright_{t_k'} c_{k+1}' \blacktriangleright_t c_{k+1}'' \blacktriangleright_{t_{k+1}'} \dots \blacktriangleright_{t_{n-1}'} c_{n-1}'' \\ c_0' \blacktriangleright_{t_0'} \dots \blacktriangleright_{t_k'} c_{k+1}' = c_{k+1}' \blacktriangleright_{t_{k+1}'} \dots \blacktriangleright_{t_{n-1}'} c_{n-1}' \tilde{\blacktriangleright}_t c \end{array}$$

It is enough to show $c = c_{n-1}''$, which follows from (1) $c \approx_t c_{n-1}''$ and (2) $c \approx_{\bar{t}} c_{n-1}''$.

The sequence $c_{k+1}' \dots c_{n-1}'$ preserves partition of t (Theorem 2, point 1) and thus $c_{k+1}' \approx_t c_{n-1}'$. We can therefore use Lemma 4 on $c_{k+1}' \blacktriangleright_t c_{k+1}''$ and $c_{n-1}' \tilde{\blacktriangleright}_t c$ obtaining: (3) $c_{k+1}'' \approx_t c$, (4) $c_{k+1}' \approx_{\bar{t}} c_{k+1}''$, and (5) $c_{n-1}' \approx_{\bar{t}} c$.

Then again by Theorem 2, point 1, the sequence $c_{k+1}'' \dots c_{n-1}''$ preserves partition of t and thus $c_{k+1}'' \approx_t c_{n-1}''$. From that and (3) we have (1) $c \approx_t c_{n-1}''$.

From (4) and Theorem 2, point 2 we get (6) $c_{n-1}' \approx_{\bar{t}} c_{n-1}''$. From (5) and (6) we get (2) $c \approx_{\bar{t}} c_{n-1}''$. □