Invariants, Modularity, and Rights*

Ernie Cohen[1], Eyad Alkassar[2], Vladimir Boyarinov[3], Markus Dahlweid[4], Ulan Degenbaev[2], Mark Hillebrand[3], Bruno Langenstein[3], Dirk Leinenbach[3], Michał Moskal[4], Steven Obua[2], Wolfgang Paul[2], Hristo Pentchev[2], Elena Petrova[2], Thomas Santen[4], Norbert Schirmer[3], Sabine Schmaltz[2], Wolfram Schulte[5], Andrey Shadrin[2], Stephan Tobies[4], Alexandra Tsyban[2], and Sergey Tverdyshev[2]

[1] Microsoft Corporation, Redmond, WA, USA
ernie.cohen@microsoft.com
[2] Saarland University, Computer Science Dept., Saarbrücken, Germany
{eyad,ulan,obua,wjp,pentchev,petrova,sabine,shadrin,azul,deru}@wjpserver.cs.uni-sb.de
[3] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
{Vladimir.Boyarinov,mah,langenstein,Dirk.Leinenbach,Norbert.Schirmer}@dfki.de
[4] European Microsoft Innovation Center, Aachen, Germany
{markus.dahlweid,michal.moskal,thomas.santen,stephan.tobies}@microsoft.com
[5] Microsoft Research, Redmond, WA, USA
schulte@microsoft.com

Abstract. The quest for modular concurrency reasoning has led to recent proposals that extend program assertions to include not just knowledge about the state, but rights to access the state. We argue that these rights are really just sugar for knowledge that certain updates preserve certain invariants.

1 Introduction

Over the years, many approaches to reasoning about concurrent systems have been proposed. At their core, most of these approaches are based on *invariants*. Invariance reasoning is conceptually simple, and compositional across concurrent composition. But invariance reasoning also has a downside: to check an update to the state, you have to check all of the invariants that the update might break. This is not usually a problem when reasoning about concurrent algorithms, where you can afford to see all of the invariants. Nor is it usually a problem when reasoning about concurrent hardware or distributed systems, where the sharing of data and invariants across components is typically static. But it is a big problem when reasoning about large concurrent programs, where sharing is dynamic, and code might break invariants that are out of scope (or, indeed, might not have even been written when the code is verified).

* 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. Work of the sixteenth author was funded by the German Research Foundation (DFG) within the program 'Quality Guarantees for Computer Systems'.

The modern attack on this problem is to strengthen the specification language to specify not only a thread's *knowledge*[6] (about the state) but also its *rights* (what it is allowed to do to the state), the combination of which we call a thread's *stuff*. For example, the stuff of a thread typically includes exclusive access to its local data (that is, right to modify the data and knowledge of its exact value), and lesser rights and knowledge about shared data (e.g., one thread might be allowed only to increase a counter and another only to decrease it; the former thread can possess knowledge about its maximum value, the latter about its minimum).

In rely-guarantee reasoning [5], the stuff in a thread is static over its lifetime. In more recent approaches, stuff can move in and out of threads, e.g., through shared objects such as resources. A procedure specification describes the stuff provided to the procedure on entry, and the stuff returned on exit (or, depending on methodology, how the stuff can change). (Fork and join are conceptually similar to procedure call and return.)

The usual way to represent stuff, typified by Concurrent Separation Logic (CSL) [7], is to describe stuff using a linear logic. This elegant approach has led to some very beautiful proofs of programs. However, it is not without drawbacks:

1. Rights and knowledge are very different things, governed by very different mathematics. Knowledge follows the rules of ordinary logic and can be freely created or destroyed, while rights have to follow some conservation principles to avoid unsoundness (e.g., from duplicating and distributing an exclusive right) or resource leakage (e.g., if you forget about your exclusive right to a chunk of memory). The introduction of linearity features into the logic produces a substantial jump in computational complexity (e.g., for separation logic see [1, 2]).
2. There are many ways to form a linear space of rights. For example, in CSL, one might use either fractional permissions or counting permissions; there are also other possibilities, such as a tree-like structuring of permissions, or permission accounting using infinitesimals (as in Chalice [6]), not to mention more expressive approaches, such as relational permissions [4]. So it seems odd to build such a commitment into the programming logic.

The point of this paper is that once we have a program logic that provides ghost state and two-state invariants (which we need anyway to do internal simulation reasoning[7]), we no longer need rights within the logic; rights can be represented as the knowledge that certain updates don't break certain invariants. (In order to make knowledge – and rights – first class, we use objects as the carriers of knowledge.) This might seem odd; since knowledge can be freely

[6] We use knowledge here in the usual sense of what the thread can deduce about the state or changes thereto, not in the sense as in logics of knowledge.

[7] To show that a program simulates some abstract specification, we make this specification a (two-state) invariant of an explicit ghost object, with a 1-state coupling invariant linking it to the concrete state. The ghost object is then updated (either implicitly or explicitly) so as to maintain these invariants.

duplicated, what makes an exclusive right exclusive? The trick is to view threads as objects. The invariant of a thread is given implicitly by the annotation one would put on the thread (i.e., a disjunction with disjuncts of the form "if control is here, then this predicate holds"). So if thread T uses its exclusive right to a variable to deduce (in its assertion) that the value of the variable has some value at some point in its execution, another thread can't change the value without breaking T's invariant. Since threads are verified without access to what code might be running in other threads, this means that exclusive rights effectively prevent modifications by other threads.

The price we pay is that to move rights around, we have to manipulate ghost state. But this is to be expected, given the complexity gap between validity checking in ordinary logic and linear logics; the use of linear logics amounts to folding these manipulations of ghost state into the programming logic. One advantage to our approach is that, by building rights on top of ghost code and invariants, the "logic" of rights can be extended by just adding more code; the substitution of code verification for metatheory is usually a good trade. Equally important, software engineers understand code and invariants, whereas they are likely to reject fancy program logics. Nevertheless, our approach is compatible with the use of fancier linear rights.

The ideas here were developed in the context of the design of the Verifying C Compiler (VCC) [3], an automatic verifier for concurrent C code. However, many of the ideas are realized somewhat differently in VCC; we point out some of these differences in the footnotes.

2 Local Invariance Reasoning

Assume a state consisting of an addressable heap (containing both real state and ghost state). When we speak of a variable, we mean a heap address, and when we speak of the value of a variable a, we mean the value $[a]$ stored at the address a in a given state. Define $[A]$, where A is a set of addresses, to be the partial (heap) map restricted to the addresses in A. On top of the state, we imagine a collection of *objects*,[8] each with a unique identifier (so that we can store object references on the heap), and each with a fixed collection of invariants (two-state predicates on the heap).[9] If o denotes an object identifier,

[8] An alternative (but essentially equivalent) approach, used in VCC, is to start with objects and fields, and to use only ownership between objects. However, we then need additional system invariants to prevent aliasing objects from existing at the same time and to make sure that there is always some existing object for each bit of memory (to prevent memory leakage). Moreover, because it should be possible to change the owner of an object without having to check the object's invariant, the ownership bit for an object has to be treated specially.

[9] Note that we are assuming a fixed grain of atomicity; all objects share the same notion of system step. This might seem inelegant, but has the enormous practical advantage of allowing invariants of different objects to be freely combined with conjunction when reasoning about the state.

define $inv(o)$ to be the conjunction of the invariants of o; intuitively, we expect each object invariant should hold across each state transition (pair of consecutive states) in every execution of the program. In two-state predicates, $old(e)$ gives the value of the (single-state) expression e in the prestate, while e gives the value in the poststate. Define $unch(e) \equiv (e = old(e))$. Define the single-state invariant $inv1(o)$ to hold in a state s iff $inv(o)$ holds across the transition from s to s (the *stuttering* transition from s). Finally, define the single-state predicate $\langle p \rangle$ ("necessarily p") to mean that every possible state transition from the current state satisfies the two-state invariant p.

To enable modular checking that a state update preserves all invariants, we introduce an ownership policy on state, as follows. We add to ghost state a map *owner* from addresses to objects[10]; if $owner(a) = o$, we say that o *owns* a. Define $span(o) \equiv [\{a \mid owner(a) = o\}]$, i.e., the span of o consists of the set of locations owned by o and the values of the heap at these locations. We require the object invariants to satisfy the following *admissibility condition*, for every object o, over every possible state transition:

$$
\begin{array}{l}
(\forall o' : old(inv1(o'))) \wedge \\
(\forall o' : unch(span(o')) \vee inv(o')) \\
\Rightarrow inv(o)
\end{array}
$$

This condition says that to check that an update preserves all invariants, we only have to check the invariants of those objects who own an updated location or who acquired or released ownership of a location, i.e., whose span has changed. It thus provides the desired modularity when checking an update: we need only to find a set of objects whose spans cover the updated data, and check the invariants of these objects (assuming that all single-state invariants hold in the prestate). Admissibility checking itself is modular – we can check admissibility of an object invariant without knowing all of the object invariants (although it usually depends on some of them). Note that admissibility requires in particular that all invariants are preserved under stuttering.

To allow objects to be created and destroyed, the heap contains for each object o a (ghost) Boolean variable $exists(o)$ that says whether that object actually exists. We think of each object invariant as implicitly containing a hypothesis that the object exists in the prestate or the poststate. In addition, o has an invariant that says that it owns $exists(o)$, whether o exists or not[11].

To allow object invariants to assert the invariants of other objects, we allow invariants to contain terms of the form $inv(o)$, as long as such terms occur only with positive polarity; this polarity constraint guarantees a consistent in-

[10] We are here assuming that *owner* is not on the heap, to avoid giving it an owner; equivalently, we could put it on the heap, making its owner a system object whose only invariant is that it owns *owner*.

[11] It is important for o to own $exists(o)$ even when it doesn't exist to avoid having to worry about breaking the invariant of the old owner of $exists(o)$ when truthifying $exists(o)$.

terpretation of which invariants hold across any state transition[12]. Let I be the conjunction of all object invariants; if $I \wedge \mathit{old}([\mathit{exists}(o)]) \Rightarrow p$ (across every possible pair of states), we say that o *claims* p.

3 Structuring Invariants

Some forms of invariants are trivially admissible. For example, an invariant of object A that can be written as a predicate on $\mathit{span}(A)$ is admissible as long as it is invariant under stuttering. More generally, any invariant of the form $\mathit{unch}(\mathit{span}(A)) \vee p$ is admissible. However, invariants that depend on data owned by other objects sometimes require help from the owning objects.

Suppose that an invariant of an object B depends on some variable a in the span of an object A. For example, A might be a lower-level object forming part of the representation of B, and a might hold some part of the abstract state of A. Typically, an update to a requires checking some condition on B, or even concurrent update to B, to avoid breaking B's invariant. Without some precaution, this will make B's invariant inadmissible. We don't want to put B's particular invariant in A, because the implementation of B is not in A's scope. (Moreover, in most cases, the particular object dependent on A is state-dependent, e.g., given by some ghost variable of A, such as its owner.)

One approach is to turn the relevant state of B into an existentially quantified variable. For example, if B's invariant is a single-state invariant p that relates $[a]$ with the value of a variable b in the span of B, we can replace B's invariant with the invariant $(\exists c : p')$, where p' is p with $[b]$ replaced by c. This approach is suitable only when we don't need to constrain updates to A, and only need to mirror them by updating b appropriately[13].

An approach that we have found more useful is to allow changes to a only when some condition on the ghost heap holds, with an invariant (in A) of the form $(\mathit{unch}([a]) \vee p)$ (which itself is necessarily admissible). For example, p might be simply $[b]$, where b is a state bit owned by B, allowing B to inhibit updates to a by keeping b false; this right can move around with ownership of b. More sophisticated predicates are also possible; for example, p might require a more complex test on the state, or even a particular simultaneous update of the state. The form that we have found most useful is where p is of the form $\mathit{inv}(B)$, which essentially requires a check of B's invariant (without saying what that invariant is) when updating a; we say that B *approves* changes to a. This automatically gives B's invariants admissible use of a. We can allow this power

[12] There are various ways to weaken the polarity constraint. For example, one can stratify the objects according to a static well-founded relation (e.g., on object types), so that the invariant of object o can use $\mathit{inv}(o')$ with negative polarity only if $o' < o$, or stratify on the basis of the time when $\mathit{exists}(o)$ becomes true.

[13] Another issue is that if other objects refer to b, replacing these references with existential loses the coherence between the instances. We have considered adding to VCC existential variables that are defined by such existential formulas, but the defining formulas of such variables have to be suitably stratified to guarantee consistency.

of approval to move around by replacing the constant B with a state expression, have multiple approvers by using a conjunction of such invariants, or approve a more restricted class of changes (e.g., changes that increase a).

The simplest case of approval arises where B claims that (under some condition) A exists. There are many ways to make such a B admissible. One is for A to keep track of such "clients" with the invariant that A isn't destroyed while this set is nonempty and that taking an object out of the set requires approval of the object. (Note that this doesn't have to be done for all objects that claim the existence of A, just for those whose admissibility cannot be established in other ways.) Because B is a full-fledged object, the existence of B can be claimed by other objects, creating a graph-like information structure. Another way to structure this is to extend ownership to objects, and to give each object an implicit invariant that its owner approves its destruction or ownership changes. Yet another is to assign a fraction in the range $(0, 1]$ to each claimant, with the invariant that these claims sum to 1, which simulates fractional permissions of CSL. These can all be mixed together in the same system.

4 Threads

Because the system state is stored on the heap, the continuation of each thread has to likewise be stored on the heap, and we think of the thread as owning the locations used to represent its continuation. (For example, in a higher-order language, we would have a location for each thread that stores its continuation.) In a standard hardware architecture, we can think of the thread owning (memory locations corresponding to) the local registers (in particular, the program counter), the register data saved in the stack frames on the control stack, and any stack memory reserved beyond the current stack top. Stack variables are owned by the thread when they are allocated and when they are released, but in between ownership might pass out of the thread; this is necessary for languages like C that allow references to stack variables to be stored in data structures.

Checking admissibility of a thread means checking that updates that don't change the span of the thread don't break its (implicit) invariants. This amounts to checking that any assertion we attach to a control location is stable under any action that preserves all invariants of updated objects. This stability is normally proved using the invariants of objects mentioned in the assertion.

The invariant of a thread (like the invariant of any object) can admissibly talk about any data the thread owns. Similarly it can talk admissibly about any data whose update is approved by the thread. Note that in contrast to other approaches, where threads can only update locations that they own exclusively, nothing logically prevents a thread from changing state owned by another thread (even its program counter). However, the possibility of such updates do not effect the verification of the potentially modified thread. Moreover, as a practical matter, threads typically don't have access to the actual invariants of other threads, so we cannot verify threads that change state owned by other threads.

5 Claims

An object that owns no interesting data can nevertheless provide useful knowledge about the state (or how the state may change), through its invariant. Useful knowledge is almost never permanent; for example, knowledge about a data structure is destroyed when the structure is torn down. Thus, the admissibility of such knowledge depends on its approving destruction of the relevant parts of the state, as described in the last section. We call such an object a *claim*.

Why would we wish to use a claim to pass information around, as opposed to an ordinary assertion within program code? The answer is that code assertions can only speak sensibly about state that is owned by the thread running the code, whereas shared objects (e.g., locks) are usually not owned by the thread. Even if some property of a shared object is known to hold at some point in a program, any write to nonlocal state can destroy such information. Verifying that such information is not destroyed typically requires using invariants of objects that are out of scope (e.g., because they are invariants of lower level data objects whose implementation is hidden). Even if the invariants are in scope, this would force the properties being maintained to be proved over and over again, which would be a disaster for practical reasoning. Conversely, the knowledge carried within a claim is guaranteed to stay around until the claim itself is destroyed; because the claim is typically owned by the thread, this can only happen if the thread itself destroys the claim. Thus, claims allows knowledge to be broken up into logical units, these units moved around as necessary (put into data structures, passed in and out of procedures, etc.).

The admissibility check when forming a claim amounts to checking that its invariant is stable (i.e., cannot be falsified) as long as the claim exists; it is essentially analogous to the check of an assertion associated with a program location, except that it cannot assume the constancy of data owned by the thread. Of course the two-state invariant of the claim must hold over the transition in which the claim is "created" (i.e., when it goes from nonexistence to existence).

It is often convenient to use claims to build new claims. In order to do this, claims themselves must keep track of these *dependent* claims, so that the dependents can approve destruction of the claim. Such claims can be destroyed only when all of its dependents have been destroyed (or are simultaneously destroyed). A program using such a claim thus has to maintain (through program assertions or object invariants) information about the possible dependents that might still exist.

Claims are often passed as ghost arguments to procedures[14]. Typically, a precondition of the procedure guarantees that the claim exists and is owned by the thread executing the procedure (so that it remains in existence until the thread destroys it or gives up its ownership). There are several possible idioms for what the procedure can do with the claim. The most usual is that the

[14] It is also possible to simply assert as a precondition the existence of a claim with the suitable properties, but passing it as a ghost argument has the advantage of immediately giving it a name to which it can be referred to in ghost code.

precondition guarantees that the claim is returned with the same dependents as upon entry[15]. In some cases, the procedure has to be able to destroy the claim (e.g., if it is destroying an object referenced by the claim)[16]; in this case, the precondition also specifies the claimants that might exist on entry.

Procedures that operate on shared synchronization objects (such as locks) typically take as a ghost argument a claim that claims that the target object exists. From these initial claims, a thread can deduce the existence of other objects (possibly claims themselves). For example, acquisition procedures typically return an object with ownership of the object transfered to the calling thread; for exclusive access (as in a writer lock) this object is the very object protected by the lock, whereas for shared access (as in a reader lock), the object is a claim claiming the existence of the protected object.

6 Permissions

We return to the question of what it means to have permission to perform an action. Suppose we want to update the heap at some location, say by atomically setting it to 0. What would justify such an update?

If the thread owns the updated location, the thread's invariant is all that has to be checked. By the form of the thread invariant, this means just checking that if performing the update from a state satisfying the program assertion preceding the update results in a state satisfying the program assertion following the update. This is just ordinary sequential program reasoning.

On the other hand, if the updated location is owned by some object, we have to check that object's invariant (as well as that of the thread). The obvious thing to do is to use the prestate to deduce which object owns the location, and that the state is such that the update preserves this object's invariant. Often this approach is possible. For example, in the code implementing a concurrent object, the procedures updating some private part of the object state usually have enough local information to do this check. The majority of atomic updates in commercial code can be checked in this way (if the hardware intrinsics are treated as primitives).

However, there are cases where this approach is insufficient. First, procedures that serve as low-level wrappers of atomic hardware intrinsics (e.g., interlocked increment,) cannot talk about all possible objects that might own (or refer to) the updated location. Second, even if code updating the heap knows the object that owns the data and can see its invariant, this object might use approval or similar mechanisms that require checking the invariants of other objects; since these other objects are typically at higher levels, their invariants are likely to be out of scope (as well they should be).

[15] This corresponds to returning the same "amount" of claim in logics based on fractional permissions.

[16] To make this more convenient, claims in VCC have the property that once destroyed they can never again be recreated, allowing the destroyer of a claim to assert that the claim doesn't exist on procedure return.

Let us consider a typical example, where an object A has an invariant $unch([a]) \vee inv(B)$. We'd like to pass to the code updating a a claim c that it can use to check the update to a. When updating a, we cannot soundly assume the invariant of c holds across the update, even if the update doesn't destroy c. However, we can safely assume that the invariant of c holds over the transition that stutters from the prestate of the update. To get from this information about a nonstuttering transition from the prestate (such as the update to a, we use a claim with a (single-state) invariant that talks about all possible transitions from the prestate. To allow the code to update a without breaking B, we pass to it a claim that claims $\langle p \Rightarrow inv(B)\rangle$. For example, if B has the invariant $[a] \leq [b]$, then from a claim that claims that $[b] = 5$ we can construct a dependent claim claiming $\langle unch(span(B)) \wedge [a] \leq 5 \Rightarrow inv(B)\rangle$, which says that any change that doesn't change B and satisfies $[a] \leq 5$ preserves the invariant of B.

Note that this technique is more modular than a rely-guarantee condition, because A might have other approvers besides B (that the client might not even know about). The claim doesn't claim that an update satisfying p will satisfy all invariants (which would be impossible without breaking information hiding), only that it will not break B's invariant.

Now, just as we don't want to expose information about B to the code, we also don't want to expose details of the update to the client providing the claim (since the update to a might need to simultaneously update other data belonging to A. All that p has to specify (beyond the change to a) is that the update doesn't update the span of B. For example, if the whole invariant of B (beyond ownership of b) is $([exists(A)] \wedge [a] < [b])$, a suitable claim would be one that claims $\langle old([a]) \geq [a] \wedge unch(span(B)) \Rightarrow inv(B)\rangle$ (which can be read as: from the current state, any state change that doesn't increase a and doesn't change B preserves the invariant of B). In general, we can view any claim of the form $\langle p\rangle$ as giving information about the effects of potential updates, and therefore a form of partial permission.

7 Read Permissions

So far, we have talked about permissions that allow a thread to change the state. Fractional permissions or counting permissions (as described in the implementation of claims) are often used in logics such as CSL to allow reading part of the state.

In the view presented here, reading a location requires no permission at all; the reason for having a read permission is to allow the thread reading the location to make a subsequent assertion about the location (such as its having the same value that was read). That is, the read permission is just an invariant that makes the subsequent assertion admissible. In the CSL tradition, a read permission specifically guarantees that the location isn't changing, which can be expressed in an ordinary invariant.

A natural objection is that this means we would be certifying programs that read possibly "invalid" regions of memory (which would, of course, result in a page fault on typical hardware). One response would be that such reads are not really "reads", but calls to lower level reading procedures that require that the memory being read is valid[17].

8 Superposition

In some cases, permission isn't enough. In some cases, b must be updated along with a, e.g., to preserve an invariant in B of the form $[a] = [b]$. Note that in real software, this situation would only arise when b was a ghost variable, whereas a could be either real or ghost. We call the required update to b that restores an invariant a *compensation*. The need for compensation creates a dilemma: we can't update b within the code that knows about a (because b is out of scope), nor in the code that knows about b (since any required updates to private parts of A would not be possible).

What we need to do is to pass a suitable compensation to the code updating a; the compensation thus looks like a callback that is called within the atomic action that updates b. This is a bit tricky, because the compensation has to "run" starting from a state (after the update of a, but still within the atomic action) where object invariants might no longer hold (not even for objects that haven't been modified). So validation of the callback usually needs to know something about the update that preceded it. Dually, the atomic action needs to know some properties of the callback.

We could pass the compensation as an explicit (mathematical) function from states to states, but since the compensation updates only ghost state, it is sufficient to know that a state representing the result of the compensation exists. So we can pass a compensation in the form of a claim that claims

$$(\forall S : p(S_0, S) \Rightarrow (\exists S' : q(S, S') \wedge r(S_0, S')))$$

where S_0 denotes the current state. Here, p describes what the caller (or whoever justifies the compensation) knows about the update, q describes what the code performing the atomic action needs to know about the compensation, and r describes what it needs to know about the combined effect. So in the case of the invariant $[a] = [b]$, we could define $p \equiv \mathit{unch}(\mathit{span}(B))$, $q \equiv \mathit{unch}(\mathit{span}(A))$, and $r \equiv \mathit{inv}(B)$. The claim can be constructed[18] by defining S' to be the state obtained by applying the update $b := a$ to the state S. Within the atomic action,

[17] In VCC, in the name of efficacy we dispense with these explicit memory access procedures, and simply keep track of which memory locations are valid according to the rules of C, checking that all memory accesses are to valid memory locations.

[18] An automatic verifier can hardly be expected to guess the witnessing Skolem function $S'(S_0, S)$ automatically, so the code constructing the compensation claim gives explicit code performing the necessary compensation, i.e., the code snippet $b := a$. Note that, like all ghost code, this code has to be guaranteed to terminate to ensure soundness, and any nondeterminism can be considered angelic rather than demonic.

the code updates a, then simply moves to an arbitrary state S' satisfying the condition given by the claim.

9 Automata

The claim used to provide permission in the last section allows an update to be done an arbitrary number of times. Sometimes, we want to allow an update to happen only once. For example, if we are simulating a step of a processor, we might in a single step write to memory while simultaneously updating the (virtual) program counter. This permission can only be used once – we don't want execution of a single machine instruction to result in multiple writes to shared memory.

We can get this effect in two ways. One is for the compensation to require the destruction of the permission as part of the atomic action. (Note that because permissions are objects, they are effectively additive – if a thread gains two permission objects, he can use them for two separate updates.) The other approach is to use a more complex form of permission that, instead of being based on claims, is based on more general objects that can own additional "local" state that is updated when the permission is used. Such an object can represent more complex permissions that allow operations to be performed only according to some (arbitrarily complex) protocol (given by the invariant of the object). Moreover, the local state can be used to make sure that the client has actually used the permission when it returns. (In the case of simulating the processor step, this allows the caller to ascertain that the virtual program counter has actually moved forward.)

10 Implementation

The development of VCC has been driven by the verification of the Microsoft Hypervisor (the core component of Hyper-V$^{\text{TM}}$) as part of the Verisoft XT project[19]. The hypervisor, consisting of 100KLOC of concurrent C and about 6KLOC of x64 assembler, runs directly on multiprocessor x64 hardware, turning it into a number of virtual multiprocessor x64 machines (with an extra level of virtual address translation, to allow each machine to be given the illusion of 0-based contiguous memory). Except for moderate size, it is fairly representative of low-level commercial system software: it contains a small operating system (albeit without devices), complete with kernel, memory manager, scheduler, debugger, etc. The most complex part of the system (which uses shadow page tables to provide a virtual TLB) uses a number of very subtle concurrent algorithms, with a quite complex simulation relation.

In VCC, most objects correspond to structured type declarations within the code. That is, for each struct declaration, we provide annotations giving its invariants; these invariants apply to each instance of the type. By default, each

[19] `http://www.verisoftxt.de`

object owns its fields (except for fields of compound types, which are considered separate objects; large structs can be broken up by introducing ghost substructures). The type declarations are proved admissible using only type information (they don't need to examine the code). Claims are treated differently from ordinary type definitions, because most claims are local to the code of a single procedure. So the admissibility of a claim is checked at the point at which the claim is formed in the code.

In VCC, there are actually two levels of object construction. The first level merely gives an object ownership of some memory; it guarantees that, in any state, the heap is interpreted in a consistent way. Whenever code accesses memory using a structured type, it requires existence of the structured object. The second level of existence is called "closing" an object; it is only while an object is closed that its declared invariants hold.

In C, there is an important difference between access to variables that are owned by a thread and those that might be concurrently accessed by another thread; in the former case, the compiler can safely reorder operations, while in the latter it cannot. In C, accesses of the later type must be marked as *volatile*, to prevent such optimizations. Only volatile data is update in explicit atomic actions; nonvolatile updates are treated using ordinary sequential reasoning. Nonvolatile fields of an object can only be updated when the object is open and owned by the updating thread.

11 Conclusion

In the world of security, the rights abstraction was introduced for a very practical reason: it provides a simple characterization of what a principal (such as a thread) might do, one that can be simply understood and can be enforced with simple hardware and software mechanisms in a small trusted computing base. It also provided a degree of modularity: a thread can check that it has the rights it needs so that it doesn't get stuck, and can keep certain rights to itself to make sure that other threads don't interfere.

The main lesson of this paper is that rights are a natural derivative of knowledge and invariance, rather than a fundamental notion. From a methodological standpoint, this is enabled by an alternative approach (admissibility) to the required modularity of invariance reasoning. In our approach, the expressiveness of rights grows naturally with the expressiveness of knowledge, and that new rights abstractions can be introduced through programming rather than through extensions to the logic and metatheory. From an implementation standpoint, it allows reuse of the substantial infrastructure built up to reason about knowledge, without the need to introduce new program logics.

These observations do not mean that expressive logics combining knowledge and rights are not a good idea; they provide useful abstractions and guidance for how proofs of programs can be structured. We have even considered including such notations in VCC, as syntactic sugar. But we are very conservative when it comes to extending the program logic itself, and our general policy is to avoid

doing so when the desired functionality can be built at the program level. We have not yet found such extensions necessary.

References

1. Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. In *CSL '08: Proceedings of the 22nd international workshop on Computer Science Logic*, volume 5213 of *Lecture Notes in Computer Science*, pages 323–338. Springer, 2008.
2. Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *APLAS*, pages 289–300, 2001.
3. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer. Invited paper.
4. Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2009.
5. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
6. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2009.
7. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.