VCC: Contract-based Modular Verification of Concurrent C

Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies
European Microsoft Innovation Center
Ritterstrasse 23, 52072 Aachen, Germany
{markus.dahlweid,michal.moskal,thomas.santen,stephan.tobies}@microsoft.com

Wolfram Schulte
Microsoft Research
Redmond, WA, USA
schulte@microsoft.com

Abstract

Most system level software is written in C and executed concurrently. Because such software is often critical for system reliability, it is an ideal target for formal verification. Annotated C and the Verified C Compiler (VCC) form the first modular sound verification methodology for concurrent C that scales to real-world production code. VCC is integrated in Microsoft Visual Studio and it comes with support for verification debugging: an explorer for counter-examples of failed proofs helps to find errors in code or specifications, and a prover log analyzer helps debugging proof attempts that exhaust available resources (memory, time). VCC is currently used to verify the core of Microsoft Hyper-V, consisting of 50,000 lines of system-level C code.

1. VCC – A C Verifier

VCC is a fully-automated tool that verifies partial correctness of annotated C code. Annotated C augments C with type invariants and pre-/postcondition-style function contracts, similar to, yet more powerful than design-by-contract assertions known from object-oriented programming languages like Eiffel.

The annotation language is classical first-order predicate logic disguised in C-like syntax, such that the annotations can be kept inline with the production code and developers can easily comprehend the annotations. Contracts specify the effect of C functions executing in a concurrent context (cf. Section 2), such that callers of a function need to know only the contract of that function to verify their own implementation. Accordingly, the verification approach is modular: a function can be verified independently of its use. This is essential to make the verification scale.

Technically, C preprocessor macros can be used to hide the annotations from an ordinary C compiler, which is used when generating the executable code. VCC translates the C program with its annotations to BoogiePL [4], a simple intermediate language for verification purposes. This translation combines features of ordinary C compilation (e.g., performing type conversions required by C) and those of a classical VC generator, e.g., by asserting type-safe memory access, flagging likely errors like arithmetic overflows, and weakening of intermediate assertions to compensate for interference from other threads. Boogie translates each function into a set of verification conditions in first-order logic and then uses the Z3 [3] automatic SMT solver to prove these formulas valid.

If Z3 can disprove a verification condition, it generates a counter-example, which consists of a sequence of program states with variable assignments. VCC's *Model Viewer* presents the counter-example to the developer. It shows the trace through the method that leads to the violation of the verification condition; in addition it shows the memory and additional attributes on the memory for each different state of this trace.

The verification task at hand is undecidable. Thus, the prover may fail by exceeding memory or time resources and not returning a definitive answer. The Z3 *Visualizer* allows to inspect prover logs to determine what the prover was trying to do while it ran out of resources. This may give valueable information as to how to help the tool succeed in proving a verification condition, e.g., by stating additional assertions that constrain the prover's search space. VCC is integrated in Microsoft Visual Studio, allowing developers to verify code from within their normal work environment while working on that code.

2. Verification Methodology

Typed memory. Typesafe languages like Java and C# are relatively verification-friendly: program state consists of a collection of objects, each with its own fields, some of which might be pointers to objects. Aliasing, which places a high burden on the theorem prover, can thus arise only through two pointers (of the same type) pointing to the same object. C deviates from this view of state in fundamental ways. First, C has no real objects; types merely give a way of interpreting chunks of memory. "Objects" can overlap arbitrarily (within the limits of object alignment). Second,

there is no distinction between objects and fields. A struct can contain another struct as a member, and a pointer can point to a member inside of another struct.

Still, a typed memory model is highly desirable because it significantly reduces both the burden on the annotator and the theorem prover. Thus, we introduce a flexible type system for C that allows one to treat the (common) case of well-typed programs efficiently, supressing many forms of trivial aliasing, while being flexible enough to allow for memory re-interpretations where this is required by the code. This allows for sound treatment of arbitrary C code. The type information is maintained in ghost state (ghost state is for specification purposes only and does not change program behavior), where we keep track of the set of "valid" typed pointers that point to the "real" objects of the state. This lays the foundation for the use of type invariants, which allows one to express program properties using the program's natural abstraction boundaries.

Invariants. As mentioned before, modular verification is the key to a methodology that scales to real-world systems. This, of course, also applies to the verification in the presence of concurrency, where modularity means thread-local reasoning. As in most approaches to thread-local verification, we start from disjoint concurrency, i.e. threads operating on disjoint portions of the state; this admits 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* shared objects, such as resources or locks.

Instead, our methodolgy is capable of verifying the implementation of lock-free data structures that are commonly used for thread synchronization, like spinlocks or reader/writer locks. These objects can then be used to control ownership transfer between threads without the need for additional primitives in our annotation language. The key to the verification of lock-free data structures is the introduction of two-state invariants that allow for the atomic modification of shared objects in a controlled fashion. Two-state invariants generalize object invariants by allowing reference to a pre-state and thus constraining atomic changes such that any two consecutive program states maintain a shared object's invariant.

When a thread has exclusive ownership of an object, it can temporarily disable the object's invariant to allow multiple changes to an object before the invariant can be re-established, a process called opening and closing of the manipulated object. Thus, a thread typically modifies a shared object by taking ownership of the object (from another object), opening it, modifying it, closing it, and putting it back somewhere (usually in its original place).

Claims. Threads can only meaningfully interact with shared state if they have some guarantees 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. Our methodology allows to reify this guarantee into *claim* objects, which, in their simplest form, are handles that prevent other objects from being opened, thus allowing the owner of the claim to rely on the claimed object's invariant. Claims can be created and destroyed dynamically, and thus allow to construct and tear down complex system configurations.

Technical detail on the memory model, ownership, and the verification of concurrent code can be found in [1, 2, 5].

3. Verification of Microsoft Hyper-V

Microsoft Hyper-V Server 2008 is Microsoft's server virtualization product. Its kernel, which we henceforth call "the hypervisor", is a thin layer of software. It sits directly on x64 hardware, turning a real MP x64 machine into a number of MP x64 *virtual machines* for OS virtualization. The code base consists of about 55KLOC of C and about 5KLOC of assembler. It is partitioned into about a dozen layers, with essentially no up-calls, and the data and functions within each layer separated into layer-private and layer-public parts.

VCC is currently used to formally verify the hypervisor code base *as is* by annotating code and pushing the annotated code through VCC. This effort is an ongoing collaboration of Microsoft and the University of Saarbrücken, Germany, which is partly funded by the German Ministry for Education and Research in the project Verisoft/XT [6].

References

[1] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. 2008. To appear.

[2] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. 2008. To appear.

[3] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary*, 2008.

[4] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Mar. 2005.

[5] S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In *12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, LNCS 5140, 2008.

[6] The Verisoft XT project. `http://www.verisoftxt.de/`, 2007.

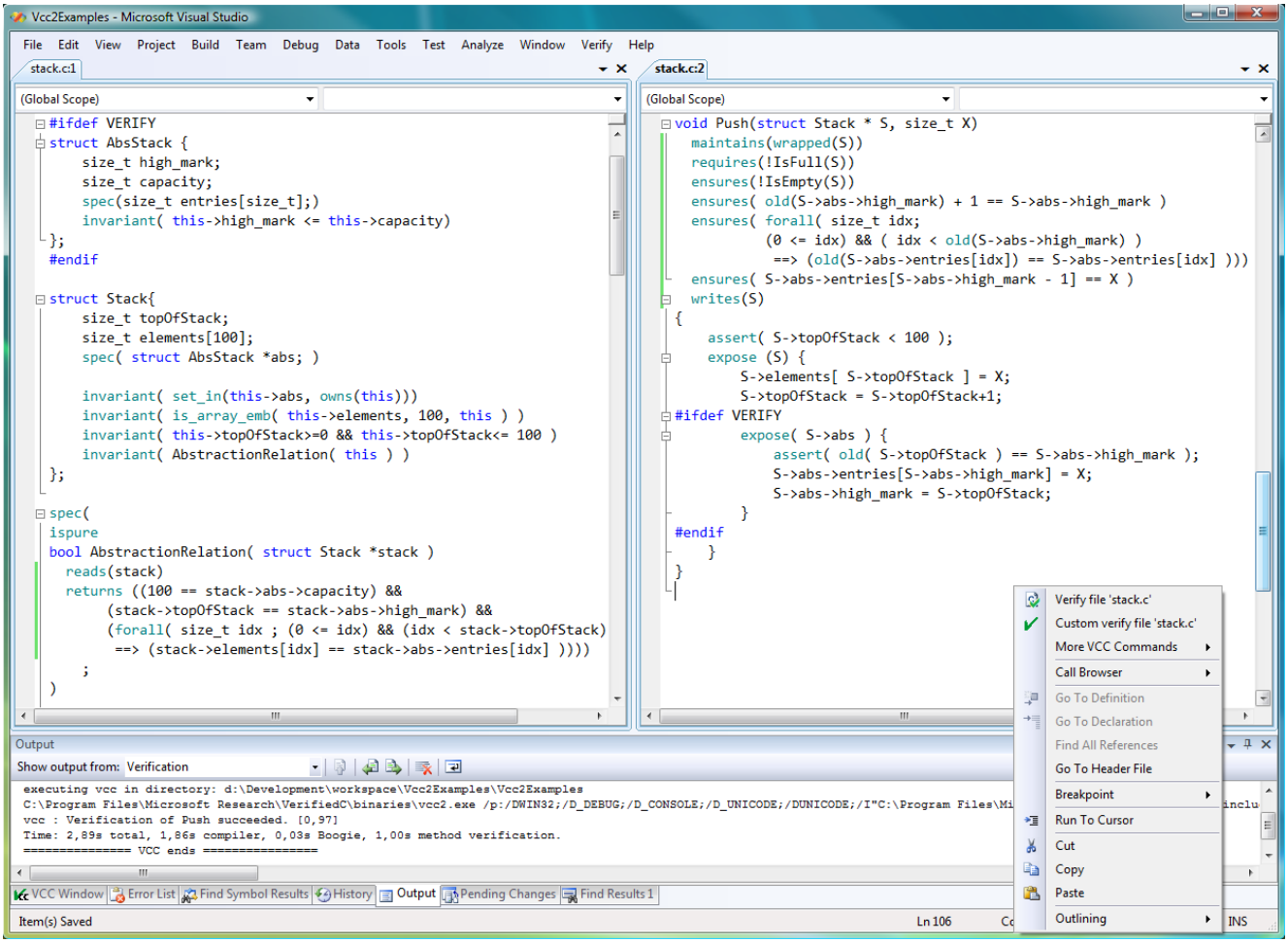

Figure 1. VCC integration into Visual Studio 2008

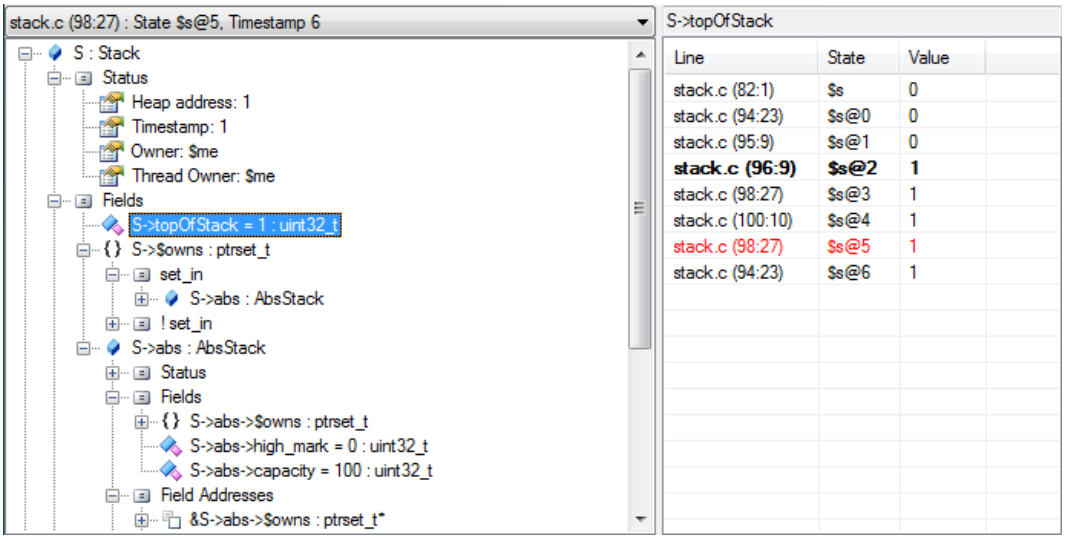

Figure 2. Visualization of Z3 error models in the VCC Model Viewer

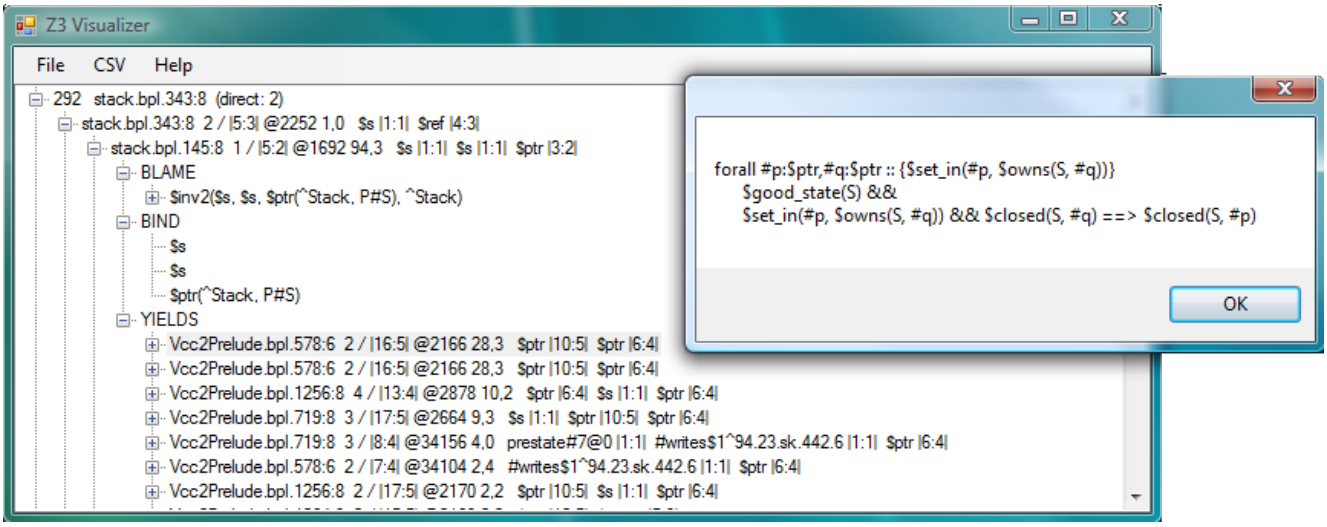

Figure 3. Z3 Visualizer